SICStus Prolog User's Manual

by the Intelligent Systems Laboratory

Swedish Institute of Computer Science
PO Box 1263
SE-164 29 Kista, Sweden

Release 3.7
September 1998

Swedish Institute of Computer Science
sicstus-request@sics.se
http://www.sics.se/isl/sicstus.html

Copyright © 1995 SICS

Swedish Institute of Computer Science
PO Box 1263
SE-164 29 Kista, Sweden

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by SICS.

Introduction

Prolog is a simple but powerful programming language developed at the University of Marseilles [Roussel 75], as a practical tool for *programming in logic* [Kowalski 74]. From a user's point of view the major attraction of the language is ease of programming. Clear, readable, concise programs can be written quickly with few errors.

For an introduction to programming in Prolog, readers are recommended to consult [Sterling & Shapiro 86]. However, for the benefit of those who do not have access to a copy of this book, and for those who have some prior knowledge of logic programming, a summary of the language is included. For a more general introduction to the field of Logic Programming see [Kowalski 79]. See Chapter 2 [Prolog Intro], page 27.

This manual describes a Prolog system developed at the Swedish Institute of Computer Science. Parts of the system were developed by the project "Industrialization of SICStus Prolog" in collaboration with Ericsson Telecom AB, NobelTech Systems AB, Infologics AB and Televerket. The system consists of a *WAM* emulator written in C, a library and runtime system written in C and Prolog and an interpreter and a compiler written in Prolog. The Prolog engine is a Warren Abstract Machine (WAM) emulator [Warren 83]. Two modes of compilation are available: in-core i.e. incremental, and file-to-file. When compiled, a predicate will run about 8 times faster and use memory more economically. Implementation details can be found in [Carlsson 90] and in several technical reports available from SICS.

SICStus Prolog follows the mainstream Prolog tradition in terms of syntax and built-in predicates, and is largely compatible with DECsystem-10 Prolog and Quintus Prolog.

Acknowledgments

The following people have contributed to the development of SICStus Prolog:

Jonas Almgren, Johan Andersson, Stefan Andersson, Kent Boortz, Per rand, Göran Båge, Mats Carlsson, Jesper Eskilson, Lena Flood, Seif Haridi, Ralph Haygood, Christian Holzbaur, Key Hyckenberg, Hans Nilsson, Mats Nylén, Greger Ottosson, Dan Sahlin, Thomas Sjöland, Johan Widén, and Emil Aström.

The OR-parallel execution model and parallelization of SICStus Prolog are due to

Khayri A.M. Ali and Roland Karlsson

The Industrialization of SICStus Prolog (1988-1991) was funded by

Ericsson Telecom AB, NobelTech Systems AB, Infologics AB and Televerket under the National Swedish Information Technology Program IT4.

The development of release 3 (1991-1995) was funded in part by

Ellemtel Utvecklings AB

This manual is based on DECsystem-10 Prolog User's Manual by

D.L. Bowen, L. Byrd, F.C.N. Pereira, L.M. Pereira, D.H.D. Warren

The Visandor visualization tool was developed at the Technical University of Madrid, and its inclusion into this distribution was kindly permitted by Manuel Hermenegildo.

See Chapter 29 [CLPQR], page 247, for acknowledgments relevant to the clp(Q,R) constraint solver.

See Chapter 30 [CLPFD], page 271, for acknowledgments relevant to the clp(FD) constraint solver.

Quintus and Quintus Prolog are trademarks of Quintus Computer Systems, Inc. UNIX is a trademark of Bell Laboratories. DEC is a trademark of Digital Equipment Corporation. Intel is a trademark of Intel Corp. Microsoft and Windows are trademarks of Microsoft Corp. OS/2 is a trademark of IBM Corp. Macintosh is a trademark of Apple Computer, Inc.

Notational Conventions

Keyboard Characters

When referring to keyboard characters, printing characters are written thus: `a`, while control characters are written like this: `^A`. Thus `^C` is the character you get by holding down the `CTL` key while you type `c`. Finally, the special control characters carriage-return, line-feed and space are often abbreviated to `RET`, `LFD` and `SPC` respectively.

Mode Spec

When introducing a built-in predicate, we shall present its usage with a *mode spec* which has the form *name(arg, ..., arg)* where each *arg* denotes how that argument should be instantiated in goals, and has one of the following forms:

:ArgName This argument should be instantiated to a term denoting a goal or a clause or a predicate name, or which otherwise needs special handling of module prefixes.

+ArgName
This argument should be instantiated to a non-variable term.

-ArgName This argument should be uninstantiated.

?ArgName
This argument may or may not be instantiated.

Mode specs are not only used in the manual, but are part of the syntax of the language as well. When used in the source code, however, the *ArgName* part must be omitted. That is, *arg* must be either `:`, `+`, `-`, or `?`.

Predicate Spec

In the context of certain compiler declarations, we shall need the following notation: Predicates in Prolog are distinguished by their name *and* their arity. The notation *name/arity* is therefore used when it is necessary to refer to a predicate unambiguously; e.g. `concatenate/3` specifies the predicate which is named "concatenate" and which takes 3 arguments. In general, a *predicate spec* takes the form *name/arity*. With the introduction of the module system, this spec is only defined relative to the "current" module. An absolute predicate spec must include a module prefix: *module*:*name/arity*.

Development and Runtime Systems

The full Prolog system with top-level, compiler, debugger etc. is known as the *Development System*. On some platforms, both a sequential and an OR-parallel Development System can be built. The OR-parallel Development System is also known as *Muse*.

OR-parallelism executes clauses and disjunctions of a predicate in parallel and is useful for search and all-solutions style programming. No change or annotations of the programs are necessary but sometimes more parallelism may be utilized if the program is modified. The main issue is to avoid side effects that might serialize the search. Except for some few exceptions Muse preserves the full functionality of SICStus Prolog. Muse is still an experimental system and is not formally supported.

It is possible to link user-written C code with a subset of SICStus Prolog to create stand-alone applications, called *Runtime Systems*. Only sequential Runtime Systems are available. When introducing a built-in predicate, any limitations on its use in Runtime Systems will be mentioned.

Function Prototypes

Whenever this manual documents a C function as part of SICStus Prolog's foreign language interface, the function prototype will be displayed in ANSI C syntax.

ISO Compliance

SICStus Prolog complies in spirit, if not in detail, with the Draft International Standard ISO/IEC 13211-1 (PROLOG: Part 1—General Core). In particular, SICStus Prolog does not offer a strictly conforming mode which rejects uses of implementation specific features. Minor deviations also concern e.g. the syntax, the arithmetic functions, and the error system.

To aid programmers who wish to write standard compliant programs, built-in predicates that have a counterpart in the ISO Prolog Standard are annotated with *[ISO]* in this manual, where appropriate with a comment clarifying the difference between the SICStus and the prescribed ISO Prolog versions.

1 How to Run Prolog

SICStus Prolog offers the user an interactive programming environment with tools for incrementally building programs, debugging programs by following their executions, and modifying parts of programs without having to start again from scratch.

The text of a Prolog program is normally created in a file or a number of files using one of the standard text editors. The Prolog interpreter can then be instructed to read in programs from these files; this is called *consulting* the file. Alternatively, the Prolog compiler can be used for *compiling* the file.

1.1 Getting Started

Under UNIX, SICStus Prolog is normally started from one of the shells. On other platforms, it is normally started by clicking on an icon. However, it is often convenient to run SICStus Prolog under GNU Emacs instead. A GNU Emacs interface for SICStus Prolog is described later (see Section 1.11 [Emacs Interface], page 18). From a shell, SICStus Prolog is started by typing:

```
% sicstus [-f] [-i] [-m] [-B[abspath]] [-R[abspath]] [-p boot-path]
    [-base executable] [-r restorefile] [-l prologfile]
    [-P[T] [num]] [-F num] [-a argument...]
```

where the flags have the following meaning:

`-f`
: Fast start. Don't read any initialization file ('`~/.sicstusrc`' or '`~/.sicstus.ini`') on startup. If the flag is omitted, SICStus Prolog will consult this file on startup if it exists.

`-i`
: Forced interactive. Prompt for user input, even if the standard input does not appear to be a terminal.

`-m`
: Use `malloc()`/`free()` in the memory manager's bottom layer.

`-l` *prolog-file*
: Ensure that the file *prolog-file* is loaded on startup.

`-r` *saved-state*
: Restore the saved state *saved-state* on startup. This is done before any initialization file or *prolog-file* is loaded.

`-a` *argument...*
: where the arguments can be retrieved from Prolog by `prolog_flag(argv,` *Args*`)`, which will unify *Args* with *argument...* represented as a list of atoms.

`-p` *boot-path*
: Overrides the default path for the Runtime Library, from which the value of the `user:library_directory/1` fact is derived. This option is only relevant for statically linked Development Systems (see Section 8.9 [Development Systems], page 156) as an alternative to setting the `SP_PATH` variable (see Section 1.1.2 [Environment Variables], page 10). The use of this flag is discouraged. Use the `-base` option instead.

`-base` *executable*
: *Unix platforms only.* Override the binary used when executing SICStus from within the start-script. Useful, for example, for statically linked Development Systems (see Section 8.9 [Development Systems], page 156) to avoid having to set environment variables such as `SP_PATH` (see Section 1.1.2 [Environment Variables], page 10) which are set by the start-script.

`-P` [*num*]
`-PT` [*num*]
: The uninstrumented (-P) or trace (-PT) version of Muse is started. As an option, *num* specifies the initial number of *workers* used (defaults to 1). See Section 3.3.1 [Muse Exe Model], page 43. Only available if Muse has been installed.

`-F` *num*
: Only useful for the Sequent Symmetry version of Muse. Its purpose is to preallocate shared memory for *num* workers. If the '`-F`' argument is not used, *num* is assumed to be equal to the default number of workers.

`-B`[*abspath*]
: Creates a saved state for a development system. This option is not needed for normal use. If *abspath* is given, it specifies the absolute pathname for the saved state. **Note:** There must not be a space before the path, or it will be interpreted as a separate option.

`-R`[*abspath*]
: Equivalent to the `-B` option, except that it builds a saved state for a runtime system instead.

Under Unix, a saved state *file* can be executed directly by typing:

```
% file argument...
```

This is equivalent to:

```
% sicstus -r file [-a argument...]
```

Note: Beginning with SICStus version 3.7, saved-states do not store the complete path of the binary `sp.exe`. Instead, they call the start-script `sicstus`, which is assumed to be found in the shell's path. If there are several versions of SICStus installed, it is up to the user to make sure that the correct start-script is found.

Notice that the flags are not available when executing saved states—all the command line arguments are treated as Prolog arguments.

The Development System responds with a message of identification and the prompt '`| ?- `' as soon as it is ready to accept input, thus:

```
SICStus 3.7: Mon Mar  2 19:22:44 MET 1998
| ?-
```

At this point the top-level is expecting input of a *directive*, i.e. a *query* or *command*. See Section 1.4 [Directives], page 12. You cannot type in clauses immediately (see Section 1.3 [Inserting Clauses],

page 12). While typing in a directive, the prompt (on following lines) becomes ' '. That is, the '`| ?- `' appears only for the first line of the directive, and subsequent lines are indented.

1.1.1 Finding Libraries at Runtime

As of SICStus Prolog 3.7, development systems and runtime systems have been reorganized around a common runtime kernel. This kernel is delivered as a shared object/DLL, which means that the runtime linker must be able to locate it at runtime. Under Windows, the runtime linker usually has no problem locating the correct DLLs. The rest of this section is therefore Unix-specific.

Most Unix systems rely partly on the so called *runtime linker search path* which tells the runtime linker where it should look for shared objects in addition to the default paths (such as '`/lib`', '`/usr/lib`', '`/usr/local/lib`', and so on).

SICStus' start-script fixes this by setting the runtime linker search path variable together with `SP_PATH` (See Section 1.1.2 [Environment Variables], page 10) to their correct values. There is also a flag `-base` to the start-script which forces it to use a different executable. This is very useful when bulding statically linked Development Systems (see Section 8.9 [Development Systems], page 156). By using the `-base` flag, the start-script can be used also for these. For example, to execute a statically linked Development System called `myapp`:

```
% sicstus -base myapp
```

However, this is only possible to do with Development Systems. Runtime Systems cannot be started using the SICStus start-script and therefore may have to set the runtime linker search path variable explicitly. The following is how a typical error looks like when the runtime linker cannot find the SICStus runtime kernel:

```
ld.so.1: /home/bob/mysicstus/lib/sicstus37beta/bin/sp.exe: fatal: \
   libsprt37.so: can't open file: errno=2
Killed
```

The cure for this is to explicitly set the runtime linker search path variable to contain the path to '`libsprt37.so`'. For `sh`, `bash`, etc. this is done by:

```
% export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/home/bob/mysicstus/lib
```

and for `csh`, `tcsh`, etc. by:

```
% setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:/home/bob/mysicstus/lib
```

The runtime linker search path variable is not always called `LD_LIBRARY_PATH`; this depends on which runtime linker the system uses and how it is configured. Other possible names are `LD_RUN_PATH`, `LIBRARY_PATH`, and `SHLIB_PATH`. The latter is mostly used under IRIX. Check your system documentation or ask your system administrator for details.

1.1.2 Environment Variables

The following environment variables can be set before starting SICStus Prolog. Some of these override the default sizes of certain areas. The sizes are given in bytes, but may be followed by `K` or `M` meaning kilobytes or megabytes respectively.

`SP_CTYPE` Selects the appropriate character set standard: The supported values are `euc` (for EUC) and `iso_8859_1` (for ISO 8859/1). The latter is the default.

`SP_PATH` This environment variable is used to determine the location of the Runtime Library. It is normally not necessary to set this environment variable; under UNIX it is hard-coded into the start-up script and under Windows it is stored in the repository.

However, under some circumstances it is necessary to set this variable explicitly, for example when executing Runtime System which do not specify a path in the third argument to `SP_initialize()`.

`TMPDIR` If set, indicates the pathname where temporary files should be created. Defaults to '`/usr/tmp`'.

`GLOBALSTKSIZE`
Governs the initial size of the global stack.

`LOCALSTKSIZE`
Governs the initial size of the local stack.

`CHOICESTKSIZE`
Governs the initial size of the choicepoint stack.

`TRAILSTKSIZE`
Governs the initial size of the trail stack.

`PROLOGINITSIZE`
Governs the size of Prolog's initial memory allocation.

`PROLOGMAXSIZE`
Defines a limit on the amount of data space which Prolog will use.

`PROLOGINCSIZE`
Governs the amount of space Prolog asks the operating system for in any given memory expansion.

`PROLOGKEEPSIZE`
Governs the size of space Prolog retains after performing some computation. By default, Prolog gets memory from the operating system as the user program executes and returns all free memory back to the operating system when the user program does not need any more. If the programmer knows that her program, once it has grown to a certain size, is likely to need as much memory for future computations, then she can advise Prolog not to return all the free memory back to the operating system by setting this variable. Only memory that is allocated above and beyond `PROLOGKEEPSIZE` is returned to the OS; the rest will be kept.

Send bug reports to `<sicstus-support@sics.se>`. Bugs tend actually to be fixed if they can be isolated, so it is in your interest to report them in such a way that they can be easily reproduced.

The mailing list `<sicstus-users@sics.se>` is a mailing list for communication among users and implementors. To [un]subscribe, write to `<sicstus-users-request@sics.se>`.

1.2 Reading in Programs

A program is made up of a sequence of clauses, possibly interspersed with directives to the system. The clauses of a predicate do not have to be immediately consecutive, but remember that their relative order may be important (see Section 2.3 [Procedural], page 34).

To input a program from a file *file*, just type the filename inside list brackets (followed by `.` and `RET`), thus:

```
| ?- [file].
```

This instructs the interpreter to read in (*consult*) the program. Note that it may be necessary to surround the whole file specification *file* with single quotes to make it a legal Prolog atom; e.g.

```
| ?- ['myfile.pl'].

| ?- ['/usr/prolog/somefile'].
```

The specified file is then read in. Clauses in the file are stored so that they can later be interpreted, while any directives are obeyed as they are encountered. When the end of the file is found, the interpreter displays on the standard error stream the time spent for read-in. This indicates the completion of the directive.

Predicates that expect the name of a Prolog source file as an argument use `absolute_file_name/2` (see Section 7.1.5 [Stream Pred], page 89) to look up the file. This predicate will first search for a file with the suffix '`.pl`' added to the name given as an argument. If this fails it will look for a file with no extra suffix added. There is also support for libraries.

In general, this directive can be any list of filenames, such as:

```
| ?- [myprog,extras,tests].
```

In this case all three files would be consulted.

The clauses for all the predicates in the consulted files will replace any existing clauses for those predicates, i.e. any such previously existing clauses in the database will be deleted.

Note that `consult/1` in SICStus Prolog behaves like `reconsult/1` in DEC-10 Prolog.

1.3 Inserting Clauses at the Terminal

Clauses may also be typed in directly at the terminal, although this is only recommended if the clauses will not be needed permanently, and are few in number. To enter clauses at the terminal, you must give the special directive:

```
| ?- [user].
|
```

and the new prompt '`|  `' shows that the system is now in a state where it expects input of clauses or directives. To return to top level, type ^D. The system responds thus:

```
{user consulted, 20 msec 200 bytes}
```

1.4 Directives: Queries and Commands

Directives are either *queries* or *commands*. Both are ways of directing the system to execute some goal or goals.

In the following, suppose that list membership has been defined by loading the following clauses from a file:

```
member(X, [X|_]).
member(X, [_|L]) :- member(X, L).
```

(Notice the use of anonymous variables written '`_`'.)

1.4.1 Queries

The full syntax of a query is '`?-`' followed by a sequence of goals. The top level expects queries. This is signaled by the initial prompt '`| ?- `'. Thus a query at top level looks like:

```
| ?- member(b, [a,b,c]).
```

Remember that Prolog terms must terminate with a full stop (`.`, possibly followed by layout text), and that therefore Prolog will not execute anything until you have typed the full stop (and then `RET`) at the end of the query.

If the goal(s) specified in a query can be satisfied, and if there are no variables as in this example, then the system answers

```
yes
```

and execution of the query terminates.

If variables are included in the query, then the final value of each variable is displayed (except for variables whose names begin with `_`). Thus the query

```
| ?- member(X, [a,b,c]).
```

would be answered by

```
X = a
```

At this point the system is waiting for input of either just a `RET` or else a `;` followed by `RET`. Simply typing `RET` terminates the query; the system responds with '`yes`'. However, typing `;` causes the system to *backtrack* (see Section 2.3 [Procedural], page 34) looking for alternative solutions. If no further solutions can be found it outputs '`no`'.

The outcome of some queries is shown below, where a number preceded by `_` is a system-generated name for a variable.

```
| ?- member(X, [tom,dick,harry]).

X = tom ;
X = dick ;
X = harry ;

no
| ?- member(X, [a,b,f(Y,c)]), member(X, [f(b,Z),d]).

X = f(b,c),
Y = b,
Z = c

yes
| ?- member(X, [f(_),g]).

X = f(_A)

yes
| ?-
```

Commands are like queries except that

1. Variable bindings are not displayed if and when the command succeeds.
2. You are not given the chance to backtrack through other solutions.

1.4.2 Commands

Commands start with the symbol '`:-`'. Any required output must be programmed explicitly; e.g. the command:

```
:- member(3, [1,2,3]), write(ok).
```

directs the system to check whether `3` belongs to the list `[1,2,3]`. Execution of a command terminates when all the goals in the command have been successfully executed. Other alternative solutions are not sought. If no solution can be found, the system prints:

{Warning: *Goal* - goal failed}

as a warning.

The principal use for commands (as opposed to queries) is to allow files to contain directives which call various predicates, but for which you do not want to have the answers printed out. In such cases you only want to call the predicates for their effect, i.e. you don't want terminal interaction in the middle of consulting the file. A useful example would be the use of a directive in a file which consults a whole list of other files, e.g.

```
:- [ bits, bobs, main, tests, data, junk ].
```

If a command like this were contained in the file '`myprog`' then typing the following at top-level would be a quick way of reading in your entire program:

```
| ?- [myprog].
```

When simply interacting with the top-level this distinction between queries and commands is not normally very important. At top-level you should just type queries normally. In a file, queries are in fact treated as commands, i.e. if you wish to execute some goals then the directive in the file must be preceded by '`:-`' or '`?-`', otherwise it would be treated as a clause.

1.5 Syntax Errors

Syntax errors are detected during reading. Each clause, directive or in general any term read in by the built-in predicate `read/1` that fails to comply with syntax requirements is displayed on the standard error stream as soon as it is read, along with its position in the input stream and a mark indicating the point in the string of symbols where the parser has failed to continue analysis, e.g.:

```
| member(X, X$L).
{SYNTAX ERROR: in line 5 (within 5-6)}
** , or ) expected in arguments **
member ( X , X
** here **
$ L ) .
```

if `$` has not been declared as an infix operator.

Note that any comments in the faulty line are not displayed with the error message. If you are in doubt about which clause was wrong you can use the `listing/1` predicate to list all the clauses which were successfully read-in, e.g.

```
| ?- listing(member/2).
```

Note: The built in predicates `read/(1-2)` normaly raise an exception on syntax errors (see Section 7.5 [Exception], page 101). The behavior is controlled by the flag `syntax_errors` (see `prolog_flag/3`).

1.6 Undefined Predicates

There is a difference between predicates that have no definition and predicates that have no clauses. The latter case is meaningful e.g. for dynamic predicates (see Section 5.2 [Declarations], page 53) that clauses are being added to or removed from. There are good reasons for treating calls to undefined predicates as errors, as such calls easily arise from typing errors.

The system can optionally catch calls to predicates that have no definition. First the user defined predicate `user:unknown_predicate_handler/3` (see Section 7.5 [Exception], page 101) is called. If undefined or if the call fails the action is governed by the state of the `unknown/2` flag which can be:

`trace` which causes calls to undefined predicates to be reported and the debugger to be entered at the earliest opportunity.

`error` which causes calls to such predicates to raise an exception (the default state). See Section 7.5 [Exception], page 101.

`fail` which causes calls to such predicates to fail.

Calls to predicates that have no clauses are not caught.

The built-in predicate `unknown(`*?OldState*`, `*?NewState*`)` unifies *OldState* with the current state and sets the state to *NewState*. The built-in predicate `debugging/0` prints the value of this state along with its other information. This state is also controlled by the flag `unknown` (see `prolog_flag/3`).

1.7 Program Execution And Interruption

Execution of a program is started by giving the system a directive which contains a call to one of the program's predicates.

Only when execution of one directive is complete does the system become ready for another directive. However, one may interrupt the normal execution of a directive by typing `^C`. This `^C` interruption has the effect of suspending the execution, and the following message is displayed:

```
Prolog interruption (h or ? for help) ?
```

At this point, the Development System accepts one-letter commands corresponding to certain actions. To execute an action simply type the corresponding character (lower or upper case) followed by `RET`. The available commands in both Development Systems are:

`a` aborts the current computation.

`c` continues the execution.

`e` exits from SICStus Prolog, closing all files.

`h`
`?` lists available commands.

The following commands are also available in the sequential Development System only:

`b` invokes a recursive top-level.

`d`
`z`
`t` switch on the debugger. See Chapter 6 [Debug Intro], page 59.

If the standard input stream is not connected to the terminal, e.g. by redirecting standard input to a file or a pipe, the above `^C` interrupt options are not available. Instead, typing `^C` causes SICStus Prolog to exit, and no terminal prompts are printed.

1.8 Exiting From The Top-Level

To exit from the top-level and return to the shell, either type `^D` at the top-level, or call the built-in predicate `halt/0`, or use the `e` (exit) command following a `^C` interruption.

1.9 Nested Executions—Break and Abort

The Prolog system provides a way to suspend the execution of your program and to enter a new incarnation of the top level where you can issue directives to solve goals etc. This is achieved by issuing the directive (see Section 1.7 [Execution], page 15):

```
| ?- break.
```

This invokes a recursive top-level, indicated by the message:

```
{ Break level 1 }
```

You can now type queries just as if you were at top-level.

If another call of `break/0` is encountered, it moves up to level 2, and so on. To close the break and resume the execution which was suspended, type `^D`. The debugger state and current input and output streams will be restored, and execution will be resumed at the predicate call where it had been suspended after printing the message:

```
{ End break }
```

Alternatively, the suspended execution can be aborted by calling the built-in predicate `abort/0`.

A suspended execution can be aborted by issuing the directive:

```
| ?- abort.
```

within a break. In this case no ^D is needed to close the break; *all* break levels are discarded and the system returns right back to top-level. I/O streams remain open, but the debugger is switched off. `abort/0` may also be called from within a program.

1.10 Saving and Restoring Program States

Once a program has been read, the system will have available all the information necessary for its execution. This information is called a *program state*.

The state of a program may be saved on disk for future execution. The state consists of all predicates and modules except built-in predicates and clauses of volatile predicates, the current operator declarations, the values of all writable Prolog flags except `debugging`, `source_info`, and the `user_*` stream aliases (see Section 7.6 [State Info], page 104), any blackboard data (see Section 7.11 [Blackboard Primitives], page 116), internal database data (see Section 7.10 [Database], page 115), and profiling data (see Section 7.15 [Profiling], page 121), but no information for source-linked debugging.

To save a program into a file *File*, type the following directive. On UNIX platforms, the file becomes executable:

```
| ?- save_program(File).
```

You can also specify a goal to be run when a saved program is restored. This is done by:

```
| ?- save_program(File, start).
```

where `start/0` is the predicate to be called.

Once a program has been saved into a file *File*, the following directive will restore the system to the saved state:

```
| ?- restore(File).
```

If a saved state has been moved or copied to another machine, the path names of foreign resources and other files needed upon restore are typically different at restore time from their save time values. To solve this problem, certain atoms will be relocated during restore as follows:

- Atoms that had '`$SP_PATH/library`' (the name of the directory containing the Prolog Library) as prefix at save time will have that prefix replaced by the corresponding restore time value.
- Atoms that had the name of the directory containing *File* as prefix at save time will have that prefix replaced by the corresponding restore time value.

The purpose of this procedure is to be able to build and deploy an application consisting of a saved state and other files as a directory tree with the saved state at the root: as long as the other files maintain their relative position in the deployed copy, they can still be found upon restore.

Note: Foreign resources, see Section 8.2 [Calling C], page 128, are unloaded by `save_program/(1,2)`. The names and paths of the resources, typically '`$SP_PATH/library`' relative, are however included in the saved state. After the save, and after restoring a saved state, this information is used to reload the foreign resources again. The state of the foreign resource in terms of global C variables and allocated memory is thus not preserved. Foreign resources may define *init* and *deinit* functions to take special action upon loading and unloading, see Section 8.2.6 [Init and Deinit Functions], page 134.

In the Muse Development System, the above predicates may only be called when the system has been adjusted to one worker; see Section 3.3.1 [Muse Exe Model], page 43.

1.11 Emacs Interface

This section explains how to use the GNU Emacs interface for SICStus Prolog, and how to customize your GNU Emacs environment for it.

The advantages of using SICStus in the Emacs environment are source-linked debugging, auto indentation, syntax highlighting, help on predefined predicates (requires the SICStus info files to be installed), loading code from inside Emacs, auto-fill mode, and more.

The mode was developed for GNU Emacs 19.34 but it works well for versions 19.31 through 20.2. Earlier versions may not be able to provide syntax highlighting. The mode has also been reported to work with XEmacs (version 19 and 20) but it will probably not work with GNU Emacs 18 or earlier.

The Emacs interface is not part of SICStus Prolog proper, but is included in the distribution for convenience. It was written by Emil Åström and Milan Zamazal, based on an earlier version of the mode written by Masanobu Umeda. Contributions has also been made by Johan Andersson, Peter Olin, Mats Carlsson, Johan Bevemyr, Stefan Andersson, and Per Danielsson (all at SICS), and Henrik Båkman at Uppsala University, Sweden. Some ideas and also a few lines of code have been borrowed (with permission) from Oz.el by Ralf Scheidhauer and Michael Mehl, the Emacs major mode for the Oz programming language. More ideas and code have been taken from the SICStus debugger mode by Per Mildner.

1.11.1 Customizing Emacs

Assuming the GNU Emacs interface for SICStus Prolog has been installed, inserting the following lines in your '`~/.emacs`' (or '`_emacs`' in case of Windows/DOS) will make Emacs use this mode automatically when editing files with a '`.pl`' extension:

```
(setq load-path (cons "/usr/local/lib/sicstus37" load-path))
(autoload 'run-prolog "prolog" "Start a Prolog sub-process." t)
(autoload 'prolog-mode "prolog" "Major mode for editing Prolog programs." t)
(setq prolog-use-sicstus-sd t)
(setq auto-mode-alist (cons '("\\.pl$" . prolog-mode) auto-mode-alist))
```

where the path in the first line is the file system path to '`prolog.el`' (the Emacs interface) and '`pltrace.el`' (the process output filter for source-linked debugging). For example, '`~/site-lisp`' means that the file is in the user's home directory, under site-lisp. MSDOS paths can be written like '`C:/Program Files/emacs-19.34/site-lisp`'.

The last line above makes sure that files ending with '`.pl`' are assumed to be Prolog files and not Perl, which is the default Emacs setting. If this is undesirable, remove that line. It is then necessary for the user to manually switch to prolog mode by typing `M-x prolog-mode` after opening a prolog file.

If the shell command `sicstus` is not available in the default path, then it is necessary to set the value of the environment variable `EPROLOG` to a shell command to invoke SICStus Prolog. This is an example for C Shell:

```
setenv EPROLOG /usr/local/bin/sicstus
```

1.11.2 Basic Configuration

If the following lines are not present in '`~/.emacs`', we suggest they are added, so that the font-lock mode (syntax coloring support) is enabled for all major modes in Emacs that support it.

```
(global-font-lock-mode t)
(setq font-lock-maximum-decoration t)
```

If one wants to add font-locking only to the prolog mode, the two lines above could be replaced by:

```
(add-hook 'prolog-mode-hook 'turn-on-font-lock)
```

If the background color of Emacs is dark and it is difficult to read the code because of the chosen colors, then it might be useful to add

```
(setq font-lock-background-mode 'dark)
```

1.11.3 Usage

A prolog process can be started by choosing `Run Prolog` from the `Prolog` menu, by typing `C-c RET`, or by typing `M-x run-prolog`. It is however not strictly necessary to start a prolog process manually since it is automatically done when consulting or compiling, if needed. The process can be restarted (i.e. the old one is killed and a new one is created) by typing `C-u C-c RET`.

Programs are run and debugged in the normal way, with terminal I/O via the `*prolog*` buffer. The most common debugging predicates are available from the menu or via key-bindings.

A particularly useful feature under the Emacs interface is source-linked debugging. This is enabled or disabled using the `Prolog/Source level debugging` menu entry. It can also be enabled by setting the Emacs variable `prolog-use-sicstus-sd` to `t` in '`~/.emacs`'. Both these methods set the Prolog flag `source_info` to `on`. The flag should be `on` while loading the code to be debugged and while debugging. If so, the debugger will display the source code location of the current goal when it prompts for a debugger command, by overlaying the beginning of the current line of code with an arrow. If `source_info` was `off` when the code was loaded, or if it was asserted or loaded from `user`, the current goal will still be shown but out of context.

Note that if the code has been modified since it was last loaded, Prolog's line number information may be invalid. If this happens, just reload the relevant buffer.

Consultation and compilation is either done via the menu or with the following key-bindings:

`C-c C-f` Consult file.

`C-c C-b` Consult buffer.

`C-c C-r` Consult region.

`C-c C-p` Consult predicate.

`C-c C-c f` Compile file.

`C-c C-c b` Compile buffer.

`C-c C-c r` Compile region.

`C-c C-c p` Compile predicate.

The boundaries used when consulting and compiling predicates are the first and last clauses of the predicate the cursor is currently in.

Other useful key-bindings are:

`M-n` Go to the next clause.

`M-p` Go to the previous clause.

`M-a` Go to beginning of clause.

`M-e` Go to end of clause.

`M-C-c` Mark clause.

`M-C-a` Go to beginning of predicate.

`M-C-e` Go to end of predicate.

`M-C-h` Mark predicate.

`M-{` Go to the previous paragraph (i.e. empty line).

`M-}` Go to the next paragraph (i.e. empty line).

`M-h` Mark paragraph.

`M-C-n` Go to matching right parenthesis.

`M-C-p` Go to matching left parenthesis.

`M-;` Creates a comment at `comment-column`. This comment will always stay at this position when the line is indented, regardless of changes in the text earlier on the line, provided that `prolog-align-comments-flag` is set to `t`.

`C-c C-t, C-u C-c C-t`
Enable and disable tracing, respectively.

`C-c C-d, C-u C-c C-d`
Enable and disable debugging, respectively.

`C-c C-z, C-u C-c C-z`
Enable and disable zipping, respectively.

`C-c C-s` Insert the *PredSpec* of the current predicate into the code.

`C-c C-n` Insert the name of the current predicate into the code. This can be useful when writing recursive predicates or predicates with several clauses. See also the `prolog-electric-dot-flag` variable below.

`C-c C-v a` Convert all variables in a region to anonymous variables. This can also be done using the `Prolog/Transform/All variables to '_'` menu entry. See also the `prolog-electric-underscore-flag` Emacs variable.

`C-c ?` Help on predicate (requires the SICStus info files to be installed), and the Emacs variable `prolog-info-predicate-index` to be set to the predicate index of the info files (defaults to `"(sicstus)Predicate Index"`).

1.11.4 Mode Line

If working with an application split into several modules, it is recommended to let the source files begin with a "mode line":

```
%%% -*- Module: ModuleName; -*-
```

The Emacs interface will look for the mode line and notify the SICStus Prolog module system that the predicates being incrementally reconsulted or recompiled belong to the module *ModuleName*. If SICStus Prolog recognizes the file as one being loaded before, it will remember what module it belongs to. If the mode line is missing, and the file has not been loaded before, the predicates will go into the type-in module. Even if the file has been loaded earlier, its filename may have a slightly different appearance to Prolog via the Emacs interface, so it is safest to always include the mode line. A mode line can be inserted by choosing `Insert/Module modeline` in the `Prolog` menu.

1.11.5 Configuration

The behavior of the Emacs interface can be controlled by a set of user-configurable settings. Some of these can be changed on the fly, while some require Emacs to be restarted. To set a variable on the fly, type `M-x set-variable RET VariableName RET Value RET`. Note that variable names can be completed by typing a few characters and then pressing `TAB`.

To set a variable so that the setting is used every time Emacs is started, add lines of the following format to '`~/.emacs`':

```
(setq VariableName Value)
```

The available settings are:

`prolog-system`
: The Prolog system to use. Available settings are `'mercury`, `'sicstus`, and `'swi`. Defaults to `'sicstus`.

`prolog-system-version`
: The version of SICStus that is used. Defaults to `'((sicstus (3 . 7)))`. Note that the spaces are significant!

`prolog-use-sicstus-sd`
: Set to `t` to enable the source-linked debugging extensions by default. Defaults to `nil`. The debugging can be enabled via the `Prolog` menu even if this variable is `nil`. Note that the source-linked debugging only works if `prolog-system-version` is set correctly.

`pltrace-port-arrow-assoc`
: Only relevant for source-linked debugging, this controls how the various ports of invocation boxes (see Section 6.1 [Procedure Box], page 59) map to arrows that point into the current line of code in source code buffers. Initialized as:

```
'(("call" . ">>>") ("exit" . "+++") ("ndexit" . "?++") ("redo" . "<<<")
  ("fail" . "---") ("exception" . "==>"))
```

  where `ndexit` is the non-deterministic variant of the Exit port.

`prolog-indent-width`
: How many positions to indent the body of a clause. Defaults to `tab-width`, normally 8.

`prolog-paren-indent`
: The number of positions to indent code inside grouping parentheses. Defaults to 4, which gives the following indentation.

```
p :-
        (   q1
        ;   q2,
            q3
        ).
```

  Note that the spaces between the parentheses and the code are automatically inserted when `TAB` is pressed at those positions.

`prolog-align-comments-flag`
: Set to `nil` to prevent single `%`-comments to be automatically aligned. Defaults to `t`.

  Note that comments with one `%` are indented to comment-column, comments with two `%` to the code level, and that comments with three `%` are never changed when indenting.

`prolog-indent-mline-comments-flag`
: Set to `nil` to prevent indentation of text inside `/* ... */` comments. Defaults `t`.

`prolog-object-end-to-0-flag`
: Set to nil to indent the closing `}` of an object definition to `prolog-indent-width`. Defaults to `t`.

`prolog-keywords`
: This is a list with keywords that are highlighted in a special color when used as commands (i.e. as `:-` *keyword*). Defaults to

```
'((sicstus ("block" "dynamic" "mode" "module" "multifile"
   "meta_predicate" "parallel" "public" "sequential" "volatile")))
```

`prolog-electric-newline-flag`
: Set to `nil` to prevent Emacs from automatically indenting the next line when pressing `RET`. Defaults to `t`.

`prolog-hungry-delete-key-flag`
: Set to `t` to enable deletion of all white space before the cursor when pressing the delete key (unless inside a comment, string, or quoted atom). Defaults to `nil`.

`prolog-electric-dot-flag`
: Set to `t` to enable the electric dot function. If enabled, pressing `.` at the end of a non-empty line inserts a dot and a newline. When pressed at the beginning of a line, a new head of the last predicate is inserted. When pressed at the end of a line with only whitespace, a recursive call to the current predicate is inserted. The function respects the arity of the predicate and inserts parentheses and the correct number of commas for separation of the arguments. Defaults to `nil`.

`prolog-electric-underscore-flag`
: Set to `t` to enable the electric underscore function. When enabled, pressing underscore (`_`) when the cursor is on a variable, replaces the variable with the anynomous variable. Defaults to `nil`.

`prolog-old-sicstus-keys-flag`
: Set to `t` to enable the key-bindings of the old Emacs interface. These bindings are not used by default since they violate GNU Emacs recommendations. Defaults to `nil`.

`prolog-use-prolog-tokenizer-flag`
: Set to `nil` to use built-in functions of Emacs for parsing the source code when indenting. This is faster than the default but does not handle some of the syntax peculiarities of Prolog. Defaults to `t`.

`prolog-parse-mode`
: What position the parsing is done from when indenting code. Two possible settings: `'beg-of-line` and `'beg-of-clause`. The first is faster but may result in erroneous indentation in `/* ... */` comments. The default is `'beg-of-line`.

`prolog-imenu-flag`
: Set to `t` to enable a new `Predicate` menu which contains all predicates of the current file. Choosing an entry in the menu moves the cursor to the start of that predicate. Defaults to `nil`.

`prolog-info-predicate-index`
: The info node for the SICStus predicate index. This is important if the online help function is to be used (by pressing `C-c ?`, or choosing the `Prolog/Help on predicate` menu entry). The default setting is `"(sicstus)Predicate Index"`.

`prolog-underscore-wordchar-flag`
: Set to `nil` to not make underscore (`_`) a word-constituent character. Defaults to `t`.

1.11.6 Tips

Some general tips and tricks for using the SICStus mode and Emacs in general are given here.

1.11.6.1 Font-locking

When editing large files, it might happen that font-locking is not done because the file is too large. Typing `M-x lazy-lock-mode` results in only the visible parts of the buffer being highlighted, which is much faster.

If the font-locking seems to be incorrect then choose `Fontify Buffer` from the `Prolog` menu.

1.11.6.2 Auto-fill mode

Auto-fill mode is enabled by typing `M-x auto-fill-mode`. This enables automatic line breaking with some features. For example, the following multiline comment was created by typing `M-;` followed by the text. The second line was indented and a `%` was added automatically.

```
dynamics([]).       % A list of pit furnace
                    % dynamic instances
```

1.11.6.3 Speed

There are several things to do if the speed of the Emacs environment is a problem:

- First of all, make sure that '`prolog.el`' and '`pltrace.el`' are compiled, i.e. that there is a '`prolog.elc`' and a '`pltrace.elc`' file at the same location as the original files. To do the compilation, start Emacs and type `M-x byte-compile-file RET Path RET`, where `Path` is the path to the '`*.el`' file. Do not be alarmed if there are a few warning messages as this is normal. If all went well, there should now be a compiled file which is used the next time Emacs is started.

- The next thing to try is changing the setting of `prolog-use-prolog-tokenizer-flag` to `nil`. This means that Emacs uses built-in functions for some of the source code parsing, thus speeding up indentation. The problem is that it does not handle all peculiarities of the Prolog syntax, so this is a trade-off between correctness and speed.
- The setting of the `prolog-parse-mode` variable also affects the speed, `'beg-of-line` being faster than `'beg-of-clause`.

1.11.6.4 Changing Colors

The prolog mode uses the default Emacs colors for font-locking as far as possible. The only custom settings are in the prolog process buffer. The default settings of the colors may not agree with your preferences, so here is how to change them.

First of all, list all available faces (a face is a combined setting of foreground and background colors, font, boldness, etc.) by typing `M-x list-faces-display`.

There are several functions that change the appearance of a face, the ones you will most likely need are:

`set-face-foreground`

`set-face-background`

`set-face-underline-p`

`make-face-bold`

`make-face-bold-italic`

`make-face-italic`

`make-face-unbold`

`make-face-unitalic`

These can be tested interactively by typing `M-x` *function-name*. You will then be asked for the name of the face to change and a value. If the buffers are not updated according to the new settings, then refontify the buffer using the `Fontify Buffer` menu entry in the `Prolog` menu.

Colors are specified by a name or by RGB values. Available color names can be listed with `M-x list-colors-display`.

To store the settings of the faces, a few lines must be added to '`~/.emacs`'. For example:

```
;; Customize font-lock faces
(add-hook 'font-lock-mode-hook
          '(lambda ()
             (set-face-foreground font-lock-variable-name-face "#00a000")
             (make-face-bold font-lock-keyword-face)
             (set-face-foreground font-lock-reference-face "Blue")
             ))
```

2 The Prolog Language

This chapter provides a brief introduction to the syntax and semantics of a certain subset of logic (*definite clauses*, also known as *Horn clauses*), and indicates how this subset forms the basis of Prolog.

2.1 Syntax, Terminology and Informal Semantics

2.1.1 Terms

The data objects of the language are called *terms*. A term is either a *constant*, a *variable* or a *compound term*.

2.1.1.1 Integers

The constants include *integers* such as

```
0   1   999   -512
```

Besides the usual decimal, or base 10, notation, integers may also be written in any base from 2 to 36, of which base 2 (binary), 8 (octal), and 16 (hex) are probably the most useful. Letters `A` through ! `Z` (upper or lower case) are used for bases greater than 10. E.g.

```
15   2'1111   8'17
```

all represent the integer fifteen.

There is also a special notation for character constants. E.g.

```
0'A   0'\x41   0'\101
```

are all equivalent to `65` (the character code for `A`). '`0'`' followed by any character except \ (backslash) is thus read as an integer. If '`0'`' is followed by \, the \ denotes the start of an *escape sequence* with special meaning (see [Escape Sequences], page 401).

2.1.1.2 Floats

Constants also include *floats* such as

```
1.0   -3.141   4.5E7   -0.12e+8   12.0e-9
```

Note that there must be a decimal point in floats written with an exponent, and that there must be at least one digit before and after the decimal point.

2.1.1.3 Atoms

Constants also include *atoms* such as

```
a   void   =   :=   'Algol-68'   []
```

Constants are definite elementary objects, and correspond to proper nouns in natural language. For reference purposes, here is a list of the possible forms which an atom may take:

1. Any sequence of alphanumeric characters (including `_`), starting with a lower case letter.
2. Any sequence from the following set of characters:
   `+-*/\^<>=`~:.?@#$&`
   This set can in fact be larger; see [Token String], page 398 for a precise definition.
3. Any sequence of characters delimited by single quotes. If the single quote character is included in the sequence it must be written twice, e.g. `'can''t'`. Backslashes in the sequence denote escape sequences (see [Escape Sequences], page 401).
4. Any of: `! ; [] {}`
   Note that the bracket pairs are special: `[]` and `{}` are atoms but `[`, `]`, `{`, and `}` are not. However, when they are used as functors (see below) the form `{`*X*`}` is allowed as an alternative to `{}(`*X*`)`. The form `[`*X*`]` is the normal notation for lists, as an alternative to `.(X,[])`.

2.1.1.4 Variables

Variables may be written as any sequence of alphanumeric characters (including `_`) starting with either a capital letter or `_`; e.g.

```
X   Value   A   A1   _3   _RESULT
```

If a variable is only referred to once in a clause, it does not need to be named and may be written as an *anonymous* variable, indicated by the underline character `_`. A clause may contain several anonymous variables; they are all read and treated as distinct variables.

A variable should be thought of as standing for some definite but unidentified object. This is analogous to the use of a pronoun in natural language. Note that a variable is not simply a writable storage location as in most programming languages; rather it is a local name for some data object, cf. the variable of pure LISP and identity declarations in Algol68.

2.1.1.5 Compound Terms

The structured data objects of the language are the compound terms. A compound term comprises a *functor* (called the principal functor of the term) and a sequence of one or more terms called *arguments*. A functor is characterized by its name, which is an atom, and its *arity* or number of arguments. For example the compound term whose functor is named `point` of arity 3, with arguments `X`, `Y` and `Z`, is written

```
point(X, Y, Z)
```

Note that an atom is considered to be a functor of arity 0.

Functors are generally analogous to common nouns in natural language. One may think of a functor as a record type and the arguments of a compound term as the fields of a record. Compound terms are usefully pictured as trees. For example, the term

```
s(np(john),vp(v(likes),np(mary)))
```

would be pictured as the structure

```
          s
        /   \
     np       vp
     |       /  \
   john     v     np
            |     |
          likes  mary
```

Sometimes it is convenient to write certain functors as operators—2-ary functors may be declared as infix operators and 1-ary functors as prefix or postfix operators. Thus it is possible to write, e.g.

```
X+Y     (P;Q)     X<Y      +X     P;
```

as optional alternatives to

```
+(X,Y)   ;(P,Q)   <(X,Y)   +(X)   ;(P)
```

The use of operators is described fully below (see Section 2.6 [Operators], page 37).

Lists form an important class of data structures in Prolog. They are essentially the same as the lists of LISP: a list either is the atom `[]` representing the empty list, or is a compound term with functor `.` and two arguments which are respectively the head and tail of the list. Thus a list of the first three natural numbers is the structure

```
  .
 / \
1    .
    / \
   2    .
       / \
      3   []
```

which could be written, using the standard syntax, as

```
.(1,.(2,.(3,[])))
```

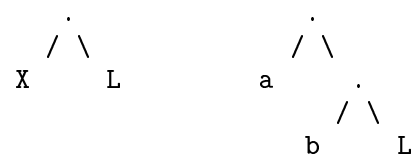
but which is normally written, in a special list notation, as

```
[1,2,3]
```

The special list notation in the case when the tail of a list is a variable is exemplified by

```
[X|L]      [a,b|L]
```

representing



respectively.

Note that this notation does not add any new power to the language; it simply makes it more readable. e.g. the above examples could equally be written

```
.(X,L)     .(a,.(b,L))
```

For convenience, a further notational variant is allowed for lists of integers which correspond to character codes. Lists written in this notation are called *strings*. E.g.

```
"SICStus"
```

which represents exactly the same list as

```
[83,73,67,83,116,117,115]
```

As for quoted atoms, if a double quote character is included in the sequence it must be written twice, e.g. `'can''t'`. Backslashes in the sequence denote escape sequences (see [Escape Sequences], page 401).

2.1.2 Programs

A fundamental unit of a logic program is the *goal* or procedure call. e.g.

```
gives(tom, apple, teacher)   reverse([1,2,3], L)   X<Y
```

A goal is merely a special kind of term, distinguished only by the context in which it appears in the program. The (principal) functor of a goal identifies what *predicate* the goal is for. It corresponds roughly to a verb in natural language, or to a procedure name in a conventional programming language.

A logic program consists simply of a sequence of statements called *sentences*, which are analogous to sentences of natural language. A sentence comprises a *head* and a *body*. The head either consists of a single goal or is empty. The body consists of a sequence of zero or more goals (i.e. it too may be empty). If the head is not empty, the sentence is called a *clause*.

If the body of a clause is empty, the clause is called a *unit clause*, and is written in the form

P.

where *P* is the head goal. We interpret this declaratively as

Goals matching *P* are true.

and procedurally as

Goals matching *P* are satisfied.

If the body of a clause is non-empty, the clause is called a *non-unit clause*, and is written in the form

P `:-` *Q*`,` *R*`,` *S*.

where *P* is the head goal and *Q*, *R* and *S* are the goals which make up the body. We can read such a clause either declaratively as

P is true if *Q* and *R* and *S* are true.

or procedurally as

To satisfy goal *P*, satisfy goals *Q*, *R* and *S*.

A sentence with an empty head is called a *directive* (see Section 1.4 [Directives], page 12), of which the most important kind is called a *query* and is written in the form

`?-` *P*`,` *Q*.

where *P* and *Q* are the goals of the body. Such a query is read declaratively as

Are *P* and *Q* true?

and procedurally as

Satisfy goals *P* and *Q*.

Sentences generally contain variables. Note that variables in different sentences are completely independent, even if they have the same name—i.e. the *lexical scope* of a variable is limited to a single sentence. Each distinct variable in a sentence should be interpreted as standing for an arbitrary entity, or value. To illustrate this, here are some examples of sentences containing variables, with possible declarative and procedural readings:

1. `employed(`*X*`) :- employs(`*Y*`,`*X*`).`

   "Any *X* is employed if any *Y* employs *X*."

   "To find whether a person *X* is employed, find whether any *Y* employs *X*."

2. `derivative(`*X*`,`*X*`,1).`

   "For any *X*, the derivative of *X* with respect to *X* is 1."

   "The goal of finding a derivative for the expression *X* with respect to *X* itself is satisfied by the result 1."

3. `?- ungulate(`*X*`), aquatic(`*X*`).`

   "Is it true, for any *X*, that *X* is an ungulate and *X* is aquatic?"

   "Find an *X* which is both an ungulate and aquatic."

In any program, the *predicate* for a particular (principal) functor is the sequence of clauses in the program whose head goals have that principal functor. For example, the predicate for a 3-ary functor `concatenate/3` might well consist of the two clauses

```
concatenate([], L, L).
concatenate([X|L1], L2, [X|L3]) :- concatenate(L1, L2, L3).
```

where `concatenate(`*L1*`,`*L2*`,`*L3*`)` means "the list *L1* concatenated with the list *L2* is the list *L3*". Note that for predicates with clauses corresponding to a base case and a recursive case, the preferred style is to write the base case clause first.

In Prolog, several predicates may have the same name but different arities. Therefore, when it is important to specify a predicate unambiguously, the form *name/arity* is used; e.g. `concatenate/3`.

Certain predicates are predefined by built-in predicates supplied by the Prolog system. Such predicates are called *built-in predicates*.

As we have seen, the goals in the body of a sentence are linked by the operator '`,`' which can be interpreted as conjunction ("and"). It is sometimes convenient to use an additional operator '`;`', standing for disjunction ("or"). (The precedence of '`;`' is such that it dominates '`,`' but is dominated by '`:-`'.) An example is the clause

```
grandfather(X, Z) :-
        (mother(X, Y); father(X, Y)),
        father(Y, Z).
```

which can be read as

> For any *X*, *Y* and *Z*, *X* has *Z* as a grandfather if either the mother of *X* is *Y* or the father of *X* is *Y*, and the father of *Y* is *Z*.

Such uses of disjunction can always be eliminated by defining an extra predicate—for instance the previous example is equivalent to

```
grandfather(X,Z) :- parent(X,Y), father(Y,Z).

parent(X,Y) :- mother(X,Y).
parent(X,Y) :- father(X,Y).
```

—and so disjunction will not be mentioned further in the following, more formal, description of the semantics of clauses.

The token '`|`', when used outside a list, is an alias for '`;`'. The aliasing is performed when terms are read in, so that

```
a :- b | c.
```

is read as if it were

```
a :- b ; c.
```

Note the double use of the '`.`' character. On the one hand it is used as a sentence terminator, while on the other it may be used in a string of symbols which make up an atom (e.g. the list functor `./2`). The rule used to disambiguate terms is that a '`.`' followed by *layout-text* is regarded as a sentence terminator (see [Token String], page 398).

2.2 Declarative Semantics

The semantics of definite clauses should be fairly clear from the informal interpretations already given. However it is useful to have a precise definition. The *declarative semantics* of definite clauses tells us which goals can be considered true according to a given program, and is defined recursively as follows.

> A goal is true if it is the head of some clause instance and each of the goals (if any) in the body of that clause instance is true, where an instance of a clause (or term) is obtained by substituting, for each of zero or more of its variables, a new term for all occurrences of the variable.

For example, if a program contains the preceding procedure for `concatenate/3`, then the declarative semantics tells us that

```
?- concatenate([a], [b], [a,b]).
```

is true, because this goal is the head of a certain instance of the first clause for `concatenate/3`, namely,

```
concatenate([a], [b], [a,b]) :- concatenate([], [b], [b]).
```

and we know that the only goal in the body of this clause instance is true, since it is an instance of the unit clause which is the second clause for `concatenate/3`.

2.3 Procedural Semantics

Note that the declarative semantics makes no reference to the sequencing of goals within the body of a clause, nor to the sequencing of clauses within a program. This sequencing information is, however, very relevant for the *procedural semantics* which Prolog gives to definite clauses. The procedural semantics defines exactly how the Prolog system will execute a goal, and the sequencing information is the means by which the Prolog programmer directs the system to execute the program in a sensible way. The effect of executing a goal is to enumerate, one by one, its true instances. Here then is an informal definition of the procedural semantics. We first illustrate the semantics by the simple query

```
?- concatenate(X, Y, [a,b]).
```

We find that it matches the head of the first clause for `concatenate/3`, with X instantiated to `[a|`*X1*`]`. The new variable *X1* is constrained by the new query produced, which contains a single recursive procedure call:

```
?- concatenate(X1, Y, [b]).
```

Again this goal matches the first clause, instantiating *X1* to `[b|`*X2*`]`, and yielding the new query:

```
?- concatenate(X2, Y, [])
```

Now the single goal will only match the second clause, instantiating both *X2* and Y to `[]`. Since there are no further goals to be executed, we have a solution

```
X = [a,b]
Y = []
```

i.e. a true instance of the original goal is

```
concatenate([a,b], [], [a,b])
```

If this solution is rejected, backtracking will generate the further solutions

```
X = [a]
Y = [b]

X = []
Y = [a,b]
```

in that order, by re-matching, against the second clause for concatenate, goals already solved once using the first clause.

Thus, in the procedural semantics, the set of clauses

```
H :- B1, ..., Bm.
H' :- B1', ..., Bm'.
...
```

are regarded as a *procedure definition* for some predicate H, and in a query

```
?- G1, ..., Gn.
```

each *Gi* is regarded as a *procedure call*. To execute a query, the system selects by its *computation rule* a goal, *Gj* say, and searches by its *search rule* a clause whose head matches *Gj*. Matching is done by the *unification* algorithm (see [Robinson 65] which computes the most general unifier, *mgu*, of *Gj* and *H*. The *mgu* is unique if it exists. If a match is found, the current query is *reduced* to a new query

```
?- (G1, ..., Gj-1, B1, ..., Bm, Gj+1, ..., Gn)mgu.
```

and a new cycle is started. The execution terminates when the empty query has been produced.

If there is no matching head for a goal, the execution *backtracks* to the most recent successful match in an attempt to find an alternative match. If such a match is found, an alternative new query is produced, and a new cycle is started.

In SICStus Prolog, as in other Prolog systems, the search rule is simple: "search forward from the beginning of the program".

The computation rule in traditional Prolog systems is also simple: "pick the leftmost goal of the current query". However, SICStus Prolog and other modern implementations have a somewhat more complex computation rule "pick the leftmost *unblocked* goal of the current query".

A goal can be blocked on one ore more uninstantiated variables, and a variable may block several goals. Thus binding a variable can cause blocked goals to become unblocked, and backtracking can cause currently unblocked goals to become blocked again. Moreover, if the current query is

```
?- G1, ..., Gj-1, Gj, Gj+1, ..., Gn.
```

where *Gj* is the first unblocked goal, and matching *Gj* against a clause head causes several blocked goals in *G1*, ..., *Gj-1* to become unblocked, then these goals may become reordered. The internal order of any two goals that were blocked on the *same* variable is retained, however.

Another consequence is that a query may be derived consisting entirely of blocked goals. Such a query is said to have *floundered*. The top-level checks for this condition. If detected, the outstanding blocked subgoals are printed on the standard error stream along with the answer substitution, to notify the user that the answer (s)he has got is really a speculative one, since it is only valid if the blocked goals can be satisfied.

A goal is blocked if certain arguments are uninstantiated and its predicate definition is annotated with a matching *block declaration* (see Section 5.2.4 [Block Declarations], page 55). Goals of certain built-in may also be blocked if their arguments are not sufficiently instantiated.

When this mechanism is used, the control structure resembles that of coroutines, suspending and resuming different threads of control. When a computation has left blocked goals behind, the situation is analogous to spawning a new suspended thread. When a blocked goal becomes unblocked, the situation is analogous to temporarily suspending the current thread and resuming the thread to which the blocked goal belongs.

2.4 Occurs-Check

It is possible, and sometimes useful, to write programs which unify a variable to a term in which that variable occurs, thus creating a cyclic term. The usual mathematical theory behind Logic Programming forbids the creation of cyclic terms, dictating that an *occurs-check* should be done each time a variable is unified with a term. Unfortunately, an occurs-check would be so expensive as to render Prolog impractical as a programming language. Thus cyclic terms may be created and may cause loops trying to print them.

SICStus Prolog mitigates the problem by its ability to unify, compare (see Section 7.3 [Term Compare], page 97), assert, and copy cyclic terms without looping. The `write_term/2` built-in predicate can optionally handle cyclic terms; see Section 7.1.3 [Term I/O], page 80. Unification with occurs-check and predicates testing (a)cyclicity are available in a library package; see Chapter 17 [Term Utilities], page 205. Other predicates usually do not handle cyclic terms well.

2.5 The Cut Symbol

Besides the sequencing of goals and clauses, Prolog provides one other very important facility for specifying control information. This is the *cut* symbol, written `!`. It is inserted in the program just like a goal, but is not to be regarded as part of the logic of the program and should be ignored as far as the declarative semantics is concerned.

The effect of the cut symbol is as follows. When first encountered as a goal, cut succeeds immediately. If backtracking should later return to the cut, the effect is to fail the *parent goal*, i.e. that goal which matched the head of the clause containing the cut, and caused the clause to be activated. In other words, the cut operation *commits* the system to all choices made since the parent goal was invoked, and causes other alternatives to be discarded. The goals thus rendered *determinate* are the parent goal itself, any goals occurring before the cut in the clause containing the cut, and any subgoals which were executed during the execution of those preceding goals.

For example:

```
member(X, [X|_]).
member(X, [_|L]) :- member(X, L).
```

This predicate can be used to test whether a given term is in a list. E.g.

```
| ?- member(b, [a,b,c]).
```

returns the answer '`yes`'. The predicate can also be used to extract elements from a list, as in

```
| ?- member(X, [d,e,f]).
```

With backtracking this will successively return each element of the list. Now suppose that the first clause had been written instead:

```
member(X, [X|_]) :- !.
```

In this case, the above call would extract only the first element of the list (`d`). On backtracking, the cut would immediately fail the whole predicate.

```
x :- p, !, q.
x :- r.
```

This is equivalent to

```
x := if p then q else r;
```

in an Algol-like language.

It should be noticed that a cut discards all the alternatives since the parent goal, even when the cut appears within a disjunction. This means that the normal method for eliminating a disjunction by defining an extra predicate cannot be applied to a disjunction containing a cut.

A proper use of the cut is usually a major difficulty for new Prolog programmers. The usual mistakes are to over-use cut, and to let cuts destroy the logic. A cut that doesn't destroy the logic is called a *green cut*; a cut that does is called a *red cut*. We would like to advise all users to follow these general rules. Also see Chapter 10 [Example Intro], page 177.

- Write each clause as a self-contained logic rule which just defines the truth of goals which match its head. Then add cuts to remove any fruitless alternative computation paths that may tie up memory.
- Cuts are usually placed right after the head, sometimes preceded by simple tests.
- Cuts are hardly ever needed in the last clause of a predicate.

2.6 Operators

Operators in Prolog are simply a *notational convenience*. For example, the expression `2+1` could also be written `+(2,1)`. This expression represents the data structure

```
   +
 /   \
2     1
```

and *not* the number 3. The addition would only be performed if the structure were passed as an argument to an appropriate predicate such as `is/2` (see Section 7.2 [Arithmetic], page 94).

The Prolog syntax caters for operators of three main kinds—*infix*, *prefix* and *postfix*. An infix operator appears between its two arguments, while a prefix operator precedes its single argument and a postfix operator is written after its single argument.

Each operator has a precedence, which is a number from 1 to 1200. The precedence is used to disambiguate expressions where the structure of the term denoted is not made explicit through the

use of parentheses. The general rule is that it is the operator with the *highest* precedence that is the principal functor. Thus if '`+`' has a higher precedence than '`/`', then

```
a+b/c      a+(b/c)
```

are equivalent and denote the term `+(a,/(b,c))`. Note that the infix form of the term `/(+(a,b),c)` must be written with explicit parentheses, i.e.

```
(a+b)/c
```

If there are two operators in the subexpression having the same highest precedence, the ambiguity must be resolved from the types of the operators. The possible types for an infix operator are

```
xfx      xfy      yfx
```

Operators of type `xfx` are not associative: it is a requirement that both of the two subexpressions which are the arguments of the operator must be of *lower* precedence than the operator itself, i.e. their principal functors must be of lower precedence, unless the subexpression is explicitly parenthesized (which gives it zero precedence).

Operators of type `xfy` are right-associative: only the first (left-hand) subexpression must be of lower precedence; the right-hand subexpression can be of the *same* precedence as the main operator. Left-associative operators (type `yfx`) are the other way around.

A functor named name is declared as an operator of type *Type* and precedence precedence by the command

```
:- op(precedence, type, name).
```

The argument name can also be a list of names of operators of the same type and precedence.

It is possible to have more than one operator of the same name, so long as they are of different kinds, i.e. infix, prefix or postfix. An operator of any kind may be redefined by a new declaration of the same kind. This applies equally to operators which are provided as standard. Declarations of all the standard operators can be found elsewhere (see [Standard Operators], page 403).

For example, the standard operators `+` and `-` are declared by

```
:- op(500, yfx, [ +, - ]).
```

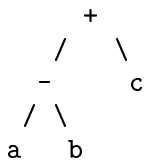
so that

```
a-b+c
```

is valid syntax, and means

```
(a-b)+c
```

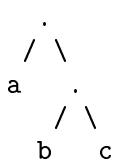
i.e.



The list functor `.` is not a standard operator, but if we declare it thus:

```
:- op(900, xfy, .).
```

then `a.b.c` would represent the structure



Contrasting this with the diagram above for `a-b+c` shows the difference between `yfx` operators where the tree grows to the left, and `xfy` operators where it grows to the right. The tree cannot grow at all for `xfx` operators; it is simply illegal to combine `xfx` operators having equal precedences in this way.

The possible types for a prefix operator are

```
fx      fy
```

and for a postfix operator they are

```
xf      yf
```

The meaning of the types should be clear by analogy with those for infix operators. As an example, if `not` were declared as a prefix operator of type `fy`, then

```
not not P
```

would be a permissible way to write `not(not(P))`. If the type were `fx`, the preceding expression would not be legal, although

```
not P
```

would still be a permissible form for `not(P)`.

If these precedence and associativity rules seem rather complex, remember that you can always use parentheses when in any doubt.

Note that the arguments of a compound term written in standard syntax must be expressions of precedence *below* 1000. Thus it is necessary to parenthesize the expression `P :- Q` in

```
| ?- assert((P :- Q)).
```

2.7 Syntax Restrictions

Note carefully the following syntax restrictions, which serve to remove potential ambiguity associated with prefix operators.

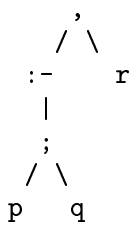
1. In a term written in standard syntax, the principal functor and its following `(` must *not* be separated by any intervening *layout-text*. Thus

   ```
   point (X,Y,Z)
   ```

   is invalid syntax.
2. If the argument of a prefix operator starts with a `(`, this `(` must be separated from the operator by at least one *layout-char*. Thus

   ```
   :-(p;q),r.
   ```

   (where '`:-`' is the prefix operator) is invalid syntax. The system would try to interpret it as the structure:

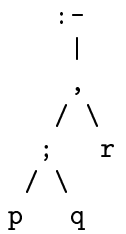
   

   That is, it would take '`:-`' to be a functor of arity 1. However, since the arguments of a functor are required to be expressions of precedence below 1000, this interpretation would fail as soon as the '`;`' (precedence 1100) was encountered.

   In contrast, the term:

   ```
   :- (p;q),r.
   ```

   is valid syntax and represents the following structure.

2.8 Comments

Comments have no effect on the execution of a program, but they are very useful for making programs more readily comprehensible. Two forms of comment are allowed in Prolog:

1. The character `%` followed by any sequence of characters up to end of line.
2. The symbol `/*` followed by any sequence of characters (including new lines) up to `*/`.

3 Running Prolog in Parallel

This chapter describes what parallel Prolog execution is about, and how Muse can be used as a "prolog accelerator" to make programs run faster in parallel than they do sequentially.

3.1 Introduction

Logic programs offer many opportunities for the exploitation of parallelism, the main ones being OR-parallelism and AND-parallelism [Conery 83]. Each kind has its own advantages and disadvantages, but we have chosen to focus on OR-parallelism as a first step for a number of reasons. From our perspective, the wide range of potential applications and the very slight execution overhead have been the main reasons. See [Lusk et al. 90] for a fuller discussion about the advantages of OR-parallelism in the context of the Aurora system.

The goal with Muse, as with Aurora, has been to allow multiple processors to exploit the *implicit* OR-parallelism of Prolog programs with a minimum of overhead, preserving the *full* Prolog language with its standard semantics.

3.2 Running a Parallel Example

In this section we illustrate the parallelism in the well-known "8-queens" problem. This code is reproduced from [Sterling & Shapiro 86] with permission from the authors:

```
% file: queens.pl

queens(N, Qs) :-
        range(1, N, Ns),
        queens(Ns, [], Qs).

queens([], Qs, Qs).
queens(UnplacedQs, SafeQs, Qs) :-
        select(UnplacedQs, UnplacedQs1, Q),
        not_attack(SafeQs, Q),
        queens(UnplacedQs1, [Q|SafeQs], Qs).

not_attack(Xs, X) :-
        not_attack(Xs, X, 1).

not_attack([], _, _) :- !.
not_attack([Y|Ys], X, N) :-
        X =\= Y+N,  X =\= Y-N,
        N1 is N+1,
        not_attack(Ys, X, N1).

select([X|Xs], Xs, X).
```

```
select([Y|Ys], [Y|Zs], X) :- select(Ys, Zs, X).

range(N, N, [N]) :- !.
range(M, N, [M|Ns]) :-
        M < N,
        M1 is M+1,
        range(M1, N, Ns).

run :-
        statistics(walltime, [Start,_]),
        findall(X, queens(10, X), L),
        statistics(walltime, [End,_]),
        length(L, N),
        Time is End - Start,
        format('~d solutions in ~3d seconds.~n', [N,Time]).
```

Note the use of the `walltime` key (i.e. real elapsed time) in the call to `statistics/2`. The `runtime` key measures CPU time on the worker that happens to be executing, and so normally is not useful.

Below we show the result of running the queens problem first with 5 workers and then with a single worker. In this case the speed-up is 3.8.

```
% sicstus -P
SICStus -P 3 #0: Mon Oct 3 12:44:31 MET 1994
| ?- compile(queens).
{compiling /src/sicstus/muse/V3/examples/queens.pl...}

yes
| ?- muse_flag(num_workers,_,5).
| ?- run.
724 solutions in 2.760 seconds.

yes
| ?- muse_flag(num_workers,_,1).
| ?- run.
724 solutions in 10.400 seconds.

yes
| ?- Speedup is 10400/2760.

Speedup = 3.7681159420289854 ?

yes
| ?- halt.

%
```

3.3 Main Issues of Parallel Execution

3.3.1 The Muse Model

The Muse model assumes a multiprocessor system with a number of processors or processes, called *workers*, with identical local address spaces, and some global address space shared by all workers. We use the term worker to represent a process or processor. The Muse model implements each worker as a sequential Prolog engine with its own WAM stacks in private memory. The Prolog program is stored in shared memory.

During execution, the actual *work* is to explore a search tree implicitly defined by the Prolog program. The available work is represented as *choicepoints* containing outstanding execution alternatives on one of the WAM stacks.

The work is performed concurrently by the workers. A distributed *scheduler* is responsible for dynamic load balancing of available work among available workers. Workers try to maximize the time they spend working and minimize the time they spend scheduling. When a worker is working, it adopts a depth-first left-to-right search strategy.

In the Muse model each worker keeps its own WAM stacks to avoid conflicting variable bindings in common parts of the search tree. Whenever a worker runs Prolog it therefore can use (more or less) the same machinery as the sequential SICStus Prolog implementation. The exceptions are (1) bookkeeping of local work each time a choicepoint is created, (2) testing for signals at predicate calls, and (3) synchronizing with other workers when taking work from a shared (public) choicepoint. The overhead for doing this is some few percent making Muse (nearly) as efficient as sequential SICStus Prolog.

When a worker (Q) runs out of work it must interrupt some busy worker (P) to get work. All of P's private choicepoints are then made public and the difference between P's and Q's state is copied from P to Q. Q can then simply simulate failure to backtrack for work.

3.3.2 Scheduling

The two main functions of the scheduler are to maintain the sequential semantics of Prolog and to match idle workers with the available work with minimal overhead. The sources of overhead in the Muse model include (1) copying a part of worker state, (2) making local choicepoints sharable, and (3) grabbing a piece of work (a task) from a shared choicepoint. Other important scheduling issues are to maximize the grain size of computations, and to prefer "promising" work over less promising work (see Section 3.3.3 [Muse Exe Cut], page 44).

3.3.3 Cut, Side Effects and Suspension

Cut has a semantics strictly compatible with sequential Prolog. Sometimes a cut may be (partly) invalid due to the fact that the current branch may be pruned by another cut of smaller scope [Hausman 90]. In this case the worker executes as much as possible of the cut operation, saves information in the public search tree on how to later on continue executing the cut operation, and then the worker continues with the current work.

Calls to predicates that produce side-effects are required to *synchronize*, i.e. to proceed only if they are on the leftmost branch of the entire tree. If a side-effect predicate is reached on a non-leftmost branch, that branch will have to suspend.

Suspending and resuming execution branches can be costly, as these actions involve copying state. The overhead can even outweigh the speedup gained from running in parallel. Therefore, the use of side-effects in parallel computations is discouraged. Also see Section 3.5 [Programming Considerations], page 45.

The current work may be endangered (i.e. possibly pruned) by some other worker executing cut or `raise_exception/1`. Such work is called *speculative work*. If there exists untaken work that is less speculative, then it might be a good idea to suspend the current work and move to the less speculative one [Ali & Karlsson 92]. This functionality (called *voluntary suspension*) is implemented in Muse and it improves execution of "search for one solution" problems dramatically, possibly from speed-up 1 to nearly linear speed-up. Unfortunately the moving of workers introduces some overhead (typically below 10 percent). Therefore it is possible to turn voluntary suspension off for programs searching for all solutions.

3.4 Visualization Tools

The distribution provides two tools that enable one to obtain insight into the *parallel* behavior of his Prolog program. These tools can help in understanding "performance bugs", situations in which a program is executing correctly but not as fast as anticipated, due to an unforeseen restriction of the degree of parallelism in the program. Once such restrictions are understood they can often be removed by changing the program, resulting in greatly improved performance.

Both of these tools use *log files*, that is, they do not display graphics while the program is running, but rather display the contents of a file that is created during program execution. This method has several advantages.

- A significant log file can be retained and studied for a long time.
- The log file can be "played" far more slowly than the original program ran, allowing careful and detailed study.
- The graphics tracing program need not run on the same machine as Muse itself.

In this manual we will not discuss the various graphics tracers in detail, but rather we will leave the details of each to its own manual. We will, however, describe how to collect the log files each one needs.

3.4.1 Must

The Must graphics tracing facility [Karlsson 92] shows the Prolog search tree as it is expanded and contracted in parallel by Muse workers.

Having created a version of Muse with the Must tracing option, the call "`muse_trace(Query)`" is used for running the query "`Query`" with tracing. After the query is run, there will be a file (with extension '`.mt`') containing a trace of the last query execution. The distribution only contains the Sun-4 executable. It can be used both under SunOS 4.x (Solaris 1.x) and SunOS 5.x (Solaris 2.x). The program itself has a dialog box to request the filename of the trace file or you can just specify the file on the command line.

3.4.2 Visandor

The Visandor graphics tracing program was originally developed for the `&-Prolog` system [Hermenegildo and Greene 90]. It is supplied in binary executable form for Sun-4 computers. More information about that tool and related work is available via WWW from: '`http://www.clip.dia.fi.upm.es`', or via email from `<clip@clip.dia.fi.upm.es>`.

The `mt2vt` tool may be used for converting a Must file to a Visandor file (with extension '`.vt`').

3.4.3 Vimust

The Must and Visandor tools can be combined into the Vimust tool. If the Must file (with extension '`.mt`') and its corresponding Visandor file (with extension '`.vt`') both exist, they can the be viewed together with the `vimust` tool.

3.5 Programming Considerations

It is possible to control the parallelism of a program by means of `parallel` and `sequential` declarations (see Section 5.2 [Declarations], page 53). Sequential declarations may be necessary to prevent infinite loops in certain programming constructs, such as failure driven loops.

Calls of interpreted predicates are synchronized and all interpreted predicates are sequential. The debugger also sequentializes the execution.

All non-deterministic built-in predicates are sequential. However, if a parallel predicate is called by `call/1`, its execution is still parallel.

We have taken pains to preserve the observable "sequential" semantics during parallel execution. This is achieved through synchronizing all built-in predicates with side-effects. As the foreign predicates may involve side-effects, in principle they should be synchronized, too. We decided not to synchronize the foreign predicates as this often may not be necessary. (The user can call the `muse_sync/0` built-in predicate immediately before the foreign predicate to achieve synchronization; see Section 7.16 [Muse Pred], page 123).

There are application areas where the principle of preserving sequential semantics seriously limits the available parallelism, since communication between parallel execution branches becomes virtually impossible. The principle may even slow down the execution, since suspending and resuming execution branches are costly operations. One way to overcome these limitations is to use the foreign language interface. As an experimental language extension, we have furthermore provided a set of primitives which can be used for non-synchronized communication. These primitives maintain a *blackboard*, i.e. a mapping from keys to terms, that can be read or written simultaneously by parallel execution branches. See Section 7.11 [Blackboard Primitives], page 116.

Warning. The blackboard predicates, as well as predicates loaded via the foreign language interface, can be executed even if the current work is speculative. These predicates are *cavalier*, i.e. a call might be executed in a parallel execution, even if that call would never be reached in a sequential execution, if the code contains cuts (see Section 2.5 [Cut], page 36) or calls to `raise_exception/1`.

This problem can be partly cured by avoiding cuts while searching for solutions, or by inserting extra validity tests in the program before calling a cavalier primitive, tests which would be redundant in a sequential execution. Remember that the validity tests may themselves succeed out of order if they contain parallelism, so they should also be written without cuts and exceptions, or be explicitly sequentialized.

Ideally, the cavalier predicates should be *asynchronous* rather than cavalier. This means that they should suspend until they are no longer endangered by cuts or calls to `raise_exception/1`, but that functionality cannot be provided in the current release, at least.

Other limitations in Muse concern certain built-in predicates and the foreign language interface. The limitations are mentioned when describing these functions. They can be summarized as follows.

- During parallel execution, predicates must not be defined (e.g. by `assert/1`) or abolished (by `abolish/(1-2)`) on the fly.
- Fastcode compilation is not yet available.
- The built-ins `stream_select/3`, `stream_interrupt/3`, and `break/0` are not available.
- Execution profiling (see Section 7.15 [Profiling], page 121) is not available.
- The built-ins `save_program/(1,2)`, `restore/1`, `load_foreign_resource/1`, `unload_foreign_resource/1`, and `load_foreign_files/2` require that Muse be adjusted to one worker.
- The `statistics/2` options `runtime` is not meaningful.
- Prolog cannot be called from C.
- The hook variable `SP_read_hook` is not available.

Muse is still an experimental system and is not formally supported.

4 The Module System

By making use of the module systems facilities, programs can be divided into different modules. Each module has its own independent predicate name space. This is an important feature for the development of larger programs. The module system of SICStus Prolog is procedure based. This means that only the predicates are local to a module, whereas terms are global. The module system is flat, not hierarchical, so all modules are visible to one another. It is non-strict, i.e. the normal visibility rules can be overridden by special syntax. No overhead is incurred on compiled calls to predicates in other modules. It is modeled after and compatible with the Quintus Prolog module system. Finally, using the module system is optional, and SICStus Prolog may be used without the user being aware of the module system at all.

Modules in SICStus Prolog can also be used for object-oriented programming. See Chapter 32 [Obj Intro], page 327, for details.

4.1 Basic Concepts

Each predicate in the Prolog system, whether built-in or user defined, belongs to a module. A predicate is generally only visible in the module where it is defined. However a predicate may be *imported* by another module. It is thereby made visible in that module too. Built-in predicates are visible in every module. Predicates declared as public in a module declaration (see below) are *exported*. Normally only public predicates may be imported by another module.

For any given goal, the *source module* is the module in which the corresponding predicate must be visible. Similarly, for any given clause, the source module of its head is the module into which the clause is loaded.

For goals occurring in a source file with a module declaration, the source module is the declared module. For goals occurring in a source file without a module declaration, the source module is the module that the file is being loaded into. For goals typed at the top level, the source module is the *type-in module*. The type-in module is by default the `user` module but may be changed by the built-in predicate `module/1`.

The other predefined module is the `prolog` module where all the built-in predicates reside. The exported built-in predicates are automatically imported into each new module as it is created.

4.2 Module Prefixing

Notwithstanding the visibility rules, any predicate can be called from any other module by prefixing the goal with the module name and the colon operator, thus overriding the source module of the goal:

```
| ?- foo:bar(X).
```

This feature is intended mainly for debugging purposes, since it defies the purposes of the module system. If the prefixed goal is a *meta-predicate*, however, the prefixed module name may affect the module name expansion of the goal (see Section 4.5 [Meta Exp], page 49).

It is also possible to override the source module of clauses and directives by module prefixing. For example,

```
:- dynamic mod:p/1.
p(X) :- mod:(q(X), r(X)).
mod:(q(X) :- r(X)).
mod:s(X) :- t(X).
```

declares `mod:p/1` as dynamic, whatever the source module is; defines `p/1` in the source module as calling `mod:q/1` and `mod:r/1`; defines `mod:q/1` as calling `mod:r/1`; and defines `mod:s/1` as calling `t/1` in the source module. The latter technique is particularly useful when the prefix is `user` and the predicate is a "hook" predicate such as `user:portray/1` which must be defined in the `user` module, but the rest of the file consists of predicates belonging to some other module.

4.3 Defining Modules

A module is normally defined by putting a module declaration first in a source file. A module declaration has the form:

`:- module(`*ModuleName*`, `*ExportList*`[, `*Options*`]).`

When the file is loaded, all predicates in the file go into *ModuleName* and the predicates of the *ExportList* are exported. When a module declaration is processed, all existing predicates in the module are erased before the new ones are loaded. A file which contains a module declaration is henceforth called a *module file*.

Options is an optional argument, and should be a list. The only available option is `hidden(`*Bool*`)`, where *Bool* is `false` (the default) or *true*. In the latter case, tracing of the predicates of the modules is disabled (although spy-points can be set), and no source information is generated at compile time.

A module can also be defined dynamically by asserting or loading predicates to it:

```
| ?- assert(m:p(x)).
```

creates the module `m`, if it does not already exists, and asserts `p(x)` to it.

```
| ?- compile(m:f).
```

creates the module `m` and loads `f` into `m`.

Dynamically created modules have no public predicates.

4.4 Importation

When a module file is loaded by `load_files/(1,2)` or one of its shorthands (see Section 7.1.1 [Read In], page 73), by default all the public predicates of the module file are imported by the receiving module. An explicit list of predicates to import may also be specified.

Clashes with already existing predicates, local or imported from other modules, are handled in two different ways: If the receiving module is the `user` module, the user is asked for redefinition of the predicate. For other receiving modules, a warning is issued and the importation is canceled. In the first case redefinition silently takes place if the flag `redefine_warnings` has the value `off` (see `prolog_flag/3`). The binding of an imported predicate remains, even if the origin is reloaded or deleted. However, `abolish/1` breaks up the importation binding. When a module file is reloaded, a check is made that the predicates imported by other modules are still in the public list. If that is not the case, a warning is issued. Note that an imported predicate may be re-exported.

4.5 Module Name Expansion

Some predicates take goals as arguments (i.e. meta-predicates). These arguments must include a module specification stating which module the goal refers. Some other predicates also need module information i.e. `compile/1`. The property of needing module information is declared with a *meta-predicate declaration* (see Section 4.6 [Meta Decl], page 50). Goals for these predicates are module name expanded to ensure the module information. Goals issued at top level and goals appearing as directives are expanded prior to execution while goals in the bodies of clauses are expanded at compile time. The expansion is made by preceding the relevant argument with '*Module*:'. If the goal is prefixed by '*Module*:', *Module* is used for the expansion, otherwise the source/type-in module is used. An argument is not expanded if:

1. It already has a module prefix.
2. It is a variable which appears in an expandable position in the head of the clause.

Some examples:

```
| ?- [user].
| :- meta_predicate p(:), q(:).
| r(X) :- p(X).
| q(X) :- p(X).
| ^D
{user consulted, 40 msec 1088 bytes}

yes
```

```
| ?- listing.

r(A) :-
        p(user:A).

q(A) :-
        p(A).

yes
```

Here, `p/1` and `q/1` are declared as meta-predicates while `r/1` is not. Thus the clause `r(X) :- p(X)` will be transformed to `r(X) :- p(`*M*`:X)`, by item 2 above, where *M* is the type-in module, whereas `q(X) :- p(X)` will not.

```
| ?- m:assert(f(1)).
```

Here, `assert/1` is called in the module `m`. However, this does not ensure that `f(1)` is asserted into `m`. The fact that `assert/1` is a meta-predicate makes the system module name expand the goal, transforming it to `m:assert(m:f(1))` before execution. This way, `assert/1` is supplied the correct module information.

4.6 Meta-Predicate Declarations

The fact that a predicate needs module name expansion is declared in a *meta-predicate declaration*:

`:- meta_predicate` *MetaPredSpec*`, ..., `*MetaPredSpec*`.`

where each *MetaPredSpec* is a mode spec. E.g.

```
:- meta_predicate p(:, +).
```

which means that the first argument of `p/2` shall be module name expanded. The arguments in the mode spec are interpreted as:

`:`
An integer
: This argument, in any call to the declared predicate, shall be expanded. (Integers are allowed for compatibility reasons).

Anything else e.g. `+`, `-` *or* `?`
: This argument shall not be expanded

A number of built-in predicates have predefined meta-predicate declarations, as indicated by the mode specs in this manual, e.g. `call(:`*Term*`)`.

5 Loading Programs

Programs can be loaded in three different ways: consulted or compiled from source file, or loaded from object files. The latter is the fastest way of loading programs, but of course requires that the programs have been compiled to object files first. Object files may be handy when developing large applications consisting of many source files, but are not strictly necessary since it is possible to save and restore entire execution states (see Section 7.17 [Misc Pred], page 124).

Consulted, or interpreted, predicates are equivalent to, but slower than, compiled ones. Although they use different representations, the two types of predicates can call each other freely.

The SICStus Prolog compiler produces compact and efficient code, running about 8 times faster than consulted code, and requiring much less runtime storage. Compiled Prolog programs are comparable in efficiency with LISP programs for the same task. However, against this, compilation itself takes about twice as long as consulting, and tracing of goals that compile in-line are not available in compiled code.

The compiler operates in three different modes, controlled by the "Compilation mode" flag (see `prolog_flag/3`). The possible states of the flag are:

`compactcode`
: Compilation produces byte-coded abstract instructions. This is the default unless SICStus Prolog has been installed with support for fastcode compilation.

`fastcode`
: Compilation produces native machine instructions. Currently only available for 68K, Sparc and MIPS platforms, but not for Muse. Fastcode runs about 3 times faster than compactcode. This is the default if SICStus Prolog has been installed with support for fastcode compilation.

`profiledcode`
: Compilation produces byte-coded abstract instructions instrumented to produce execution profiling data. See Section 7.15 [Profiling], page 121. Profiling is not available in Runtime Systems.

`debugcode`
: Compilation produces *interpreted* code, i.e. compiling is replaced by consulting.

The compilation mode can be changed by issuing the directive:

```
| ?- prolog_flag(compiling, OldValue, NewValue).
```

A Prolog program consists of a sequence of *sentences* (see [Sentence], page 396). Commands and queries encountered among the sentences are executed immediately as they are encountered, unless they can be interpreted as *declarations* (see Section 5.2 [Declarations], page 53), which affect the treatment of forthcoming clauses. Clauses are loaded as they are encountered.

A Prolog program may also contain a list of sentences (including the empty list). This is treated as equivalent to those sentences occurring in place of the list. This feature makes it possible to have `user:term_expansion/(2,4)` (see Section 7.1.2 [Definite], page 76) "return" a list of sentences, instead of a single sentence.

5.1 Predicates which Load Code

To consult a program, issue the directive:

```
| ?- consult(Files).
```

where *Files* is either the name of a file (including the file '`user`') or a list of filenames instructs the processor to read-in the program which is in the files. For example:

```
| ?- consult([dbase,'extras.pl',user]).
```

When a directive is read it is immediately executed. Any predicate defined in the files erases any clauses for that predicate already present. If the old clauses were loaded from a different file than the present one, the user will be queried first whether (s)he really wants the new definition. However, if a `multifile` declaration (see Section 5.2 [Declarations], page 53) is read and the corresponding predicate exists and has previously been declared as `multifile`, new clauses will be added to the predicate, rather than replacing the old clauses. If clauses for some predicate appear in more than one file, the later set will effectively overwrite the earlier set. The division of the program into separate files does not imply any module structure—any predicate can call any other (see Chapter 4 [Module Intro], page 47).

`consult/1`, used in conjunction with `save_program/(1,2)` and `restore/1`, makes it possible to amend a program without having to restart from scratch and consult all the files which make up the program. The consulted file is normally a temporary "patch" file containing only the amended predicate(s). Note that it is possible to call `consult(user)` and then enter a patch directly on the terminal (ending with ^D). This is only recommended for small, tentative patches.

```
| ?- [File|Files].
```

This is a shorthand way of consulting a list of files. (The case where there is just one filename in the list was described earlier (see Section 1.2 [Reading In], page 11).

To compile a program in-core, use the built-in predicate:

```
| ?- compile(Files).
```

where *Files* is specified just as for `consult/1`.

The effect of `compile/1` is very much like that of `consult/1`, except all new procedures will be stored in compiled rather than consulted form. However, predicates declared as dynamic (see below) will be stored in consulted form, even though `compile/1` is used.

To compile a program into an object file, use the built-in predicate:

```
| ?- fcompile(Files).
```

where *Files* is specified just as for `consult/1`. For each filename in the list, the compiler will append the suffix '`.pl`' to it and try to locate a source file with that name and compile it to an object file. The object filename if formed by appending the suffix '`.ql`' to the specified name.

The internal state of SICStus Prolog is not changed as result of the compilation. See Section 5.3 [Considerations], page 58.

To load a program from a set of object files, use the built-in predicate:

```
| ?- load(Files).
```

where *Files* is either a single object filename (specified with or without a trailing '`.ql`') or a list of filenames. For each filename in the list, this predicate will search for a file, with the suffix '`.ql`' added if not specified. This directive has the same effect as if the source files had been compiled using `compile/1` directly (but see Section 5.3 [Considerations], page 58).

Finally, to ensure that some files have been compiled or loaded, use the built-in predicate:

```
| ?- ensure_loaded(Files).
```

where *Files* is either a single filename or a list of filenames, similar to the arguments accepted by the above predicates. The predicate takes the following action for each *File* in the list of filenames:

1. If the *File* is `user`, `compile(user)` is performed;
2. If *File* cannot be found, not even with a '`.pl`' or '`.ql`' extension, an error is signaled;
3. If an object file is found which has not yet been loaded or which has a modification time more recent than what was recorded for the file when it was previously loaded, the file is loaded;
4. If a source file is found which has not yet been loaded or which has a modification time more recent than what was recorded for the file when it was previously loaded, the file is compiled;
5. If both a source file and an object file are found, item 3 or 4 applies depending on which file was modified most recently; in case 4 `ensure_loaded/1` does *not* cause object files to become recompiled.
6. Otherwise, no action is taken.

5.2 Declarations

When a program is to be loaded, it is sometimes necessary to tell the system to treat some of the predicates specially. This information is supplied by including *declarations* about such predicates in the source file, preceding any clauses for the predicates which they concern. A declaration is written just as a command, beginning with '`:-`'. A declaration is effective from its occurrence through the end of file.

Although declarations that affect more than one predicate may be collapsed into a single declaration, the recommended style is to write the declarations for a predicate immediately before its first clause.

Operator declarations are not declarations proper, but rather commands that modify the global table of syntax operators. Operator declarations are executed as they are encountered while loading programs.

The rest of this section details the available forms of predicate declarations.

5.2.1 Multifile Declarations

A declaration

```
:- multifile PredSpec, ..., PredSpec.
```

where each *PredSpec* is a predicate spec, causes the specified predicates to become *multifile*. This means that if more clauses are subsequently loaded from other files for the same predicate, then the new clauses will not replace the old ones, but will be added at the end instead. As of release 3, multifile declarations are required in all files from where clauses to a multifile predicate are loaded.

An example when multifile declarations are particularly useful is in defining *hook predicates*. A hook predicate is a user-defined predicate that somehow alters or customizes the behavior of SICStus Prolog. A number of such hook predicates are described in this manual. Often, an application needs to combine the functionality of several software modules, some of which define clauses for such hook predicates. By simply declaring every hook predicates as multifile, the functionality of the clauses for the hook predicates is automatically combined. If this is not done, the last software module to define clauses for a particular hook predicate will effectively supersede any clauses defined for the same hook predicate in a previous module.

If a file containing clauses for a multifile predicate is reloaded, the old clauses from the same file are removed. The new clauses are added at the end.

If a multifile predicate is loaded from a file with no multifile declaration for it, the predicate is redefined as if it were an ordinary predicate (i.e. the user is asked for confirmation).

Clauses of multifile predicates are (currently) always loaded in interpreted form, even if they were processed by the compiler. If performance is an issue, define the multifile predicates as unit clauses or as clauses with a single goal that just calls an auxiliary compiled predicate to perform any time-critical computation.

If a multifile predicate is declared *dynamic* in one file, it must also be done so in the other files from where it is loaded. Hook predicates should always be declared as multifile and dynamic, as this is the convention followed in the library modules.

Multifile declarations *must precede* any other declarations for the same predicate(s)!

5.2.2 Dynamic Declarations

A declaration

```
:- dynamic PredSpec, ..., PredSpec.
```

where each *PredSpec* is a predicate spec, causes the specified predicates to become *dynamic*, which means that other predicates may inspect and modify them, adding or deleting individual clauses. Dynamic predicates are always stored in consulted form even if a compilation is in progress. This declaration is meaningful even if the file contains no clauses for a specified predicate—the effect is then to define a dynamic predicate with no clauses.

5.2.3 Volatile Declarations

A declaration

```
:- volatile PredSpec, ..., PredSpec.
```

where each *PredSpec* is a predicate spec, causes the specified predicates to become *volatile*.

A predicate should be declared as volatile if it refers to data that cannot or should not be saved in a saved state. In most cases a volatile predicate will be dynamic, and it will be used to keep facts about streams or memory references. When a program state is saved at run-time, the clauses of all volatile predicates will be left unsaved. The predicate definitions will be saved though, which means that the predicates will keep all properties, that is *volatile* and maybe *dynamic* or *multifile*, when the saved state is restored.

5.2.4 Block Declarations

The declaration

```
:- block BlockSpec, ..., BlockSpec.
```

where each *BlockSpec* is a mode spec, specifies conditions for blocking goals of the predicate referred to by the mode spec (`f/3` say). When a goal for `f/3` is to be executed, the mode specs are interpreted as conditions for blocking the goal, and if at least one condition evaluates to `true`, the goal is blocked.

A block condition evaluates to `true` iff all arguments specified as '`-`' are uninstantiated, in which case the goal is blocked until at least one of those variables is instantiated. If several conditions evaluate to `true`, the implementation picks one of them and blocks the goal accordingly.

The recommended style is to write the block declarations in front of the source code of the predicate they refer to. Indeed, they are part of the source code of the predicate, and must precede the first clause. For example, with the definition:

```
:- block merge(-,?,-), merge(?,-,-).

merge([], Y, Y).
merge(X, [], X).
merge([H|X], [E|Y], [H|Z]) :- H @< E,  merge(X, [E|Y], Z).
merge([H|X], [E|Y], [E|Z]) :- H @>= E, merge([H|X], Y, Z).
```

calls to `merge/3` having uninstantiated arguments in the first *and* third position *or* in the second *and* third position will suspend.

The behavior of blocking goals for a given predicate on uninstantiated arguments cannot be switched off, except by abolishing or redefining the predicate.

Block declarations generalize the "wait declarations" of earlier versions of SICStus Prolog. A declaration '`:- wait f/3`' in the old syntax corresponds to '`:- block f(-,?,?)`' in the current

syntax. See Section 10.5.6 [Use Of Term Exp], page 182, for a simple way to extend the system to accept the old syntax.

5.2.5 Meta-Predicate Declarations

A declaration

```
:- meta_predicate MetaPredSpec, ..., MetaPredSpec.
```

where each *MetaPredSpec* is a mode spec, informs the compiler that certain arguments of the declared predicates are used for passing goals. To ensure the correct semantics in the context of multiple modules, clauses or directives containing goals for the declared predicates may need to have those arguments module name expanded. See Section 4.5 [Meta Exp], page 49, for details.

5.2.6 Module Declarations

A declaration

```
:- module(ModuleName, ExportList[, Options]).
```

where *ExportList* is a list of predicate specs, declares that the forthcoming predicates should go into the module named *ModuleName* and that the predicates listed should be exported. See Section 4.3 [Def Modules], page 48, for details.

5.2.7 Public Declarations

A declaration

```
:- public PredSpec, ..., PredSpec.
```

where each *PredSpec* is a predicate spec, has no effect whatsoever, but is accepted for compatibility reasons. In some Prologs, this declaration is necessary for making compiled predicates visible. In SICStus Prolog, predicate visibility is handled by the module system. See Chapter 4 [Module Intro], page 47.

5.2.8 Mode Declarations

A declaration

```
:- mode ModeSpec, ..., ModeSpec.
```

where each *ModeSpec* is a mode spec, has no effect whatsoever, but is accepted for compatibility reasons. In some Prologs, this declaration helps reduce the size of the compiled code for a predicate, and may speed up its execution. Unfortunately, writing mode declarations can be error-prone, and since errors in mode declaration do not show up while running the predicates interpretively, new

bugs may show up when predicates are compiled. However, mode declarations may be used as a commenting device, as they express the programmer's intention of data flow in predicates.

5.2.9 Parallel and Sequential Declarations

The following declarations serve for the explicit control of parallelism in the Muse Development System.

`:- parallel` *PredSpec*`, ..., ` *PredSpec*`.`

enables the clauses of the specified predicates to be run in parallel (this is the default).

`:- sequential` *PredSpec*`, ..., ` *PredSpec*`.`

prohibits the clauses of the specified predicates from running in parallel.

Universal versions of both declarations also exist:

```
:- sequential.
:- parallel.
```

This indicates to the compiler that all subsequent predicates in the given file become sequential or parallel, unless specifically declared otherwise. The default is to treat all predicates as parallel. Sequential declarations may be necessary to prevent infinite loops in certain programming constructs, such as failure driven loops. e.g. the loop

```
program :-
      initialize,
      generate_solution(X),
      process_solution(X),
      fail_if_not_last(X),
      !,
      shut_down.
```

will generate an unbounded number of suspended branches if `generate_solution/1` is a parallel predicate with an unbounded number of alternatives and `process_solution/1` involves a synchronized operation.

In some rare cases it might also be useful to prevent "slow down" due to too fine granularity in some predicate call. To find such cases you may use the Must trace tool (see Section 3.4.1 [Must], page 45).

5.3 Considerations for File-To-File Compilation

When compiling a source file to an object file, remember that clauses are loaded and directives are executed at *run time*, not at compile time. Only predicate declarations are processed at compile time. For instance, it does not work to include operator declarations or clauses of `user:term_expansion/(2,4)` or `user:goal_expansion/3` or any auxiliary predicates that they might need, and rely on the new transformations to be effective for subsequent clauses of the same file or subsequent files of the same compilation.

Any directives or clauses that affect the compile-time environment must be loaded prior to compiling source files to object files. This also holds for meta-predicates called by the source files but defined elsewhere, for module name expansion to work correctly. If this separation into files is unnatural or inconvenient, one can easily ensure that the compile-time environment is up to date by doing:

```
| ?- ensure_loaded(Files), fcompile(Files).
```

Since module name expansion takes place at compile time, the module into which the file is to be loaded must be known when compiling to object files. This is no problem for module files because the module name is picked from the module declaration. When non-module files are compiled, the file name may be prefixed with the module name that is to be used for expansion:

```
| ?- fcompile(Module:Files).
```

If an object file is loaded into a different module from which it was compiled for, a warning is issued.

6 Debugging

This chapter describes the debugging facilities that are available in Development Systems. The purpose of these facilities is to provide information concerning the control flow of your program.

In the Muse Development System, the debugger effectively sequentializes the execution. Other tools (see Section 3.4 [Visualization Tools], page 44) must be used to debug the parallel aspects of execution.

The main features of the debugging package are as follows:

- The *Procedure Box* model of Prolog execution which provides a simple way of visualizing control flow, especially during backtracking. Control flow is viewed at the predicate level, rather than at the level of individual clauses.
- The ability to exhaustively trace your program or to selectively set *spy-points*. Spy-points allow you to nominate interesting predicates at which the program is to pause so that you can interact.
- The wide choice of control and information options available during debugging.

The Procedure Box model of execution is also called the Byrd Box model after its inventor, Lawrence Byrd.

Much of the information in this chapter is also in Chapter eight of [Clocksin & Mellish 81] which is recommended as an introduction.

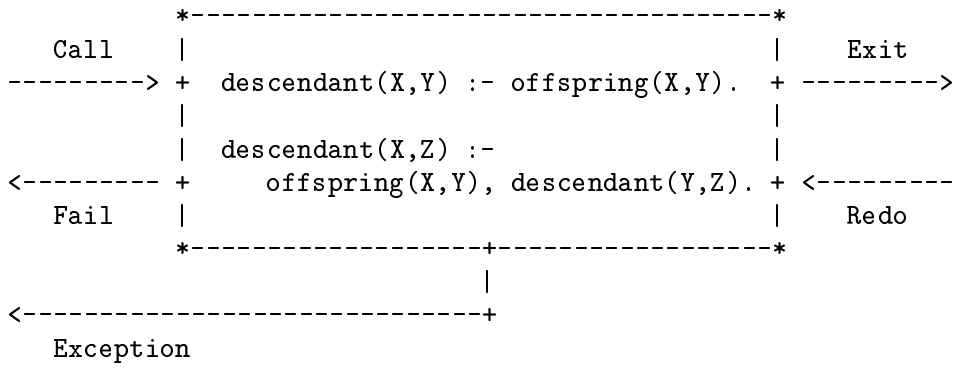
Unless otherwise stated, the debugger print goals using `write_term/3` with the value of the Prolog flag `debugger_print_options` (see Section 7.6 [State Info], page 104).

The debugger is not available in Runtime Systems and the predicates defined in this chapter are undefined; see Section 8.8 [Runtime Systems], page 151.

6.1 The Procedure Box Control Flow Model

During debugging, the debugger prints out a sequence of goals in various states of instantiation in order to show the state the program has reached in its execution. However, in order to understand what is occurring it is necessary to understand when and why the debugger prints out goals. As in other programming languages, key points of interest are predicate entry and return, but in Prolog there is the additional complexity of backtracking. One of the major confusions that novice Prolog programmers have to face is the question of what actually happens when a goal fails and the system suddenly starts backtracking. The Procedure Box model of Prolog execution views program control flow in terms of movement about the program text. This model provides a basis for the debugging mechanism in Development Systems, and enables the user to view the behavior of the program in a consistent way.

Let us look at an example Prolog predicate :



The first clause states that Y is a descendant of X if Y is an offspring of X, and the second clause states that Z is a descendant of X if Y is an offspring of X and if Z is a descendant of Y. In the diagram a box has been drawn around the whole predicate and labeled arrows indicate the control flow in and out of this box. There are five such arrows which we shall look at in turn.

Call This arrow represents initial invocation of the predicate. When a goal of the form `descendant(X,Y)` is required to be satisfied, control passes through the *Call* port of the descendant box with the intention of matching a component clause and then satisfying any subgoals in the body of that clause. Note that this is independent of whether such a match is possible; i.e. first the box is called, and then the attempt to match takes place. Textually we can imagine moving to the code for descendant when meeting a call to descendant in some other part of the code.

Exit This arrow represents a successful return from the predicate. This occurs when the initial goal has been unified with one of the component clauses and any subgoals have been satisfied. Control now passes out of the *Exit* port of the descendant box. Textually we stop following the code for descendant and go back to the place we came from.

Redo This arrow indicates that a subsequent goal has failed and that the system is backtracking in an attempt to find alternatives to previous solutions. Control passes through the *Redo* port of the descendant box. An attempt will now be made to resatisfy one of the component subgoals in the body of the clause that last succeeded; or, if that fails, to completely rematch the original goal with an alternative clause and then try to satisfy any subgoals in the body of this new clause. Textually we follow the code backwards up the way we came looking for new ways of succeeding, possibly dropping down on to another clause and following that if necessary.

Fail This arrow represents a failure of the initial goal, which might occur if no clause is matched, or if subgoals are never satisfied, or if any solution produced is always rejected by later processing. Control now passes out of the *Fail* port of the descendant box and the system continues to backtrack. Textually we move back to the code which called this predicate and keep moving backwards up the code looking for choice points.

Exception This arrow represents an exception which was raised in the initial goal, either by a call to `raise_exception/1` or by an error in a built-in predicate. See Section 7.5 [Exception], page 101. Control now passes out of the *Exception* port of the descendant box and the systems continues to pass the exception to outer levels. Textually we move

back to the code which called this predicate and keep moving backwards up the code looking for a call to `on_exception/3`.

In terms of this model, the information we get about the procedure box is only the control flow through these five ports. This means that at this level we are not concerned with which clause matches, and how any subgoals are satisfied, but rather we only wish to know the initial goal and the final outcome. However, it can be seen that whenever we are trying to satisfy subgoals, what we are actually doing is passing through the ports of *their* respective boxes. If we were to follow this, then we would have complete information about the control flow inside the procedure box.

Note that the box we have drawn round the predicate should really be seen as an *invocation box*. That is, there will be a different box for each different invocation of the predicate. Obviously, with something like a recursive predicate, there will be many different *Calls* and *Exits* in the control flow, but these will be for different invocations. Since this might get confusing each invocation box is given a unique integer identifier.

6.2 Basic Debugging Predicates

Development Systems provide a range of built-in predicates for control of the debugging facilities. The most basic predicates are as follows:

`debug` Switches the debugger on, and ensures that the next time control reaches a spy-point, a message will be produced and you will be prompted for a command. In order for the full range of control flow information to be available it is necessary to have this on from the start. When it is off the system does not remember invocations that are being executed. (This is because it is expensive and not required for normal running of programs.) You can switch *Debug Mode* on in the middle of execution, either from within your program or after a `^C` (see `trace/0` below), but information prior to this will just be unavailable.

`zip` Same as `debug/0`, except no debugging information is being collected, and so is almost as fast as running with the debugger switched off.

`trace` Switches the debugger on, and ensures that the next time control enters an invocation box, a message will be produced and you will be prompted for a command. The effect of trace can also be achieved by typing `t` after a `^C` interruption of a program.

At this point you have a number of options. See Section 6.5 [Debug Options], page 65. In particular, you can just type `RET` to *creep* (or single-step) into your program. If you continue to creep through your program you will see every entry and exit to/from every invocation box, including compiled code, except for code belonging to hidden modules (see Section 4.3 [Def Modules], page 48). You will notice that the debugger stops at all ports. However, if this is not what you want, the following built-in predicate gives full control over the ports at which you are prompted:

`leash(+`*Mode*`)`
: Leashing Mode is set to *Mode*. Leashing Mode determines the ports of invocation boxes at which you are to be prompted when you Creep through your program. At unleashed ports a tracing message is still output, but program execution does not stop to allow user interaction. Note that the ports of spy-points are always leashed (and cannot be unleashed). *Mode* can be a subset of the following, specified as a list:

  `call` Prompt on Call.

  `exit` Prompt on Exit.

  `redo` Prompt on Redo.

  `fail` Prompt on Fail.

  `exception`
  Prompt on Exception.

  The initial value of *Leashing Mode* is `[call,exit,redo,fail,exception]` (full leashing).

`nodebug`
`notrace`
`nozip` Switches the debugger off. If there are any spy-points set then they will be kept but disabled.

`debugging`
: Prints information about the current debugging state. This will show:

  1. Whether undefined predicates are being trapped.
  2. Whether the debugger is switched on.
  3. What spy-points have been set (see below).
  4. What mode of leashing is in force (see below).

6.3 Spy-points

For programs of any size, it is clearly impractical to creep through the entire program. *Spy-points* make it possible to stop the program whenever it gets to a particular predicate which is of interest. Once there, one can set further spy-points in order to catch the control flow a bit further on, or one can start creeping.

Setting a spy-point on a predicate indicates that you wish to see all control flow through the various ports of its invocation boxes, except during skips. When control passes through any port of a invocation box with a spy-point set on it, a message is output and the user is asked to interact. Note that the current mode of leashing does not affect spy-points: user interaction is requested on *every* port.

Spy-points are set and removed by the following built-in predicates. The first two are also standard operators:

`spy` *:Spec* Sets spy-points on all the predicates given by *Spec*, which is is of the form:

Name all predicates called *Name* no matter what arity, where *Name* is an atom for a specific name or a variable for all names, or

Name/Arity
the predicate of that name and arity, or

Name/Low-High
Name/(Low-High)
the predicates of that name with arity in the range *Low-High*, or

Module:Spec
specifying a particular module (see Section 4.2 [Module Spec], page 48) instead of the default type-in module, where *Module* is an atom for a specific module or a variable for all modules, or

[Spec,...,Spec]
the set of all predicates covered by the *Specs*.

You cannot place a spy-point on an undefined predicate. If you set some spy-points when the debugger is switched off then it will be automatically switched on. Examples:

```
| ?- spy [user:p, m:q/(2-3)].
| ?- spy m:[p/1, q/1].
```

`nospy` *:Spec*
Similar to `spy` *Spec* except that all the predicates given by *Spec* will have previously set spy-points removed from them.

`nospyall` Removes all the spy-points that have been set.

`spypoint_condition(`*:Goal*`,`*?Port*`,+`*Test*`)`
Sets a *conditional* spy-point on the predicate for *Goal*. When the debugger reaches the spy-point, three conditions must be met for the spy-point to apply: The actual goal must match *Goal*, the actual port must match *Port*, and *Test* (an arbitrary Prolog goal) must succeed.

The options available when you arrive at a spy-point are described later. See Section 6.5 [Debug Options], page 65.

6.4 Format of Debugging messages

We shall now look at the exact format of the message output by the system at a port. All trace messages are output to the standard error stream. This allows you to trace programs while they are performing file I/O. The basic format is as follows:

E N S `23 6 Call:` *T* `foo(hello,there,_123) ?`

E is only used at Exception ports and displays a pending exception.

N is only used at Exit ports and indicates whether the invocation could backtrack and find alternative solutions. Unintended non-determinism is a source of inefficiency, and this annotation can

help spot such efficiency bugs. It is printed as '`?`', indicating that `foo/3` could backtrack and find alternative solutions, or ' ' otherwise.

S is a spy-point indicator. It is printed as '`+`', indicating that there is a spy-point on `foo/3`, or ' ', denoting no spy-point.

The first number is the unique invocation identifier. It is nondecreasing regardless of whether or not you are actually seeing the invocations (provided that the debugger is switched on). This number can be used to cross correlate the trace messages for the various ports, since it is unique for every invocation. It will also give an indication of the number of procedure calls made since the start of the execution. The invocation counter starts again for every fresh execution of a command, and it is also reset when retries (see later) are performed.

The number following this is the *current depth*; i.e. the number of direct *ancestors* this goal has.

The next word specifies the particular port (Call, Exit, Redo, Fail, or Exception).

T is a subterm trace. This is used in conjunction with the '`^`' command (set subterm), described below. If a subterm has been selected, *T* is printed as the sequence of commands used to select the subterm. Normally, however, *T* is printed as ' ', indicating that no subterm has been selected.

The goal is then printed so that you can inspect its current instantiation state.

The final '`?`' is the prompt indicating that you should type in one of the option codes allowed (see Section 6.5 [Debug Options], page 65). If this particular port is unleashed then you will obviously not get this prompt since you have specified that you do not wish to interact at this point.

Note that calls that are compiled in-line and built-in predicates that are called directly from the top level are not traced.

There are two exceptions to the above debugger message format. A message

```
S -  -  Block: p(_133)
```

indicates that the debugger has encountered a *blocked* goal, i.e. one which is temporarily suspended due to insufficiently instantiated arguments (see Section 2.3 [Procedural], page 34). No interaction takes place at this point, and the debugger simply proceeds to the next goal in the execution stream. The suspended goal will be eligible for execution once the blocking condition ceases to exist, at which time a message

```
S -  -  Unblock: p(_133)
```

is printed.

6.5 Options Available during Debugging

This section describes the particular options that are available when the system prompts you after printing out a debugging message. All the options are one letter mnemonics, some of which can be optionally followed by a decimal integer. They are read from the standard input stream with any blanks being completely ignored up to the end of the line (`RET`). Some options only actually require the terminator; e.g. the creep option, as we have already seen, only requires `RET`.

The only option which you really have to remember is '`h`' (followed by `RET`). This provides help in the form of the following list of available options.

```
RET    creep            c      creep
 l     leap             z      zip
 s     skip             s <i>  skip i
 o     out              o <n>  out n
 q     q-skip           q <i>  q-skip i
 r     retry            r <i>  retry i
 f     fail             f <i>  fail i
 d     display          w      write
 p     print            p <i>  print partial
 g     ancestors        g <n>  ancestors n
 t     backtrace        t <n>  backtrace n
 &     blocked goals    & <n>  nth blocked goal
 n     nodebug          =      debugging
 +     spy this         + <i>  spy conditionally
 -     nospy this       .      find this
 a     abort            b      break
 @     command          u      unify
 e     raise exception
 <     reset printdepth < <n>  set printdepth
 ^     reset subterm    ^ <n>  set subterm
 ?     help             h      help
```

`c`
`RET` *creep* causes the debugger to single-step to the very next port and print a message. Then if the port is leashed (see Section 6.2 [Basic Debug], page 61), the user is prompted for further interaction. Otherwise, it continues creeping. If leashing is off, creep is the same as *leap* (see below) except that a complete trace is printed on the standard error stream.

`l` *leap* causes the debugger to resume running your program, only stopping when a spy-point is reached (or when the program terminates). Leaping can thus be used to follow the execution at a higher level than exhaustive tracing. All you need to do is to set spy-points on an evenly spread set of pertinent predicates, and then follow the control flow through these by leaping from one to the other. Debugging information is collected while leaping, so when a spy-points is reached, it is possible to inspect the ancestor goals, or creep into them upon entry to Redo ports.

`z` *zip* is like *leap*, except no debugging information is being collected while zipping, resulting in significant savings in memory and execution time.

`s` *skip* is only valid for Call and Redo ports. It skips over the entire execution of the predicate. That is, you will not see anything until control comes back to this predicate (at either the Exit port or the Fail port). Skip is particularly useful while creeping since it guarantees that control will be returned after the (possibly complex) execution within the box. If you skip then no message at all will appear until control returns. This includes calls to predicates with spy-points set; they will be masked out during the skip. No debugging information is being collected while skipping.

If you supply an integer argument, then this should denote an invocation number of an ancestral goal. The system tries to get you to the Exit or Fail port of the invocation box you have specified.

`o` *out* is a shorthand for skipping to the Exit or Fail port of the immediate ancestor goal. If you supply an integer argument *n*, it denotes skipping to the Exit or Fail port of the *n*th ancestor goal.

`q` *quasi-skip* is like a combination of *leap* and *skip*: execution stops when either control comes back to this predicate, or a spy-point is reached. No debugging information is being collected while quasi-skipping.

An integer argument can be supplied as for *skip*.

`r` *retry* can be used at any of the four ports (although at the Call port it has no effect). It transfers control back to the Call port of the box. This allows you to restart an invocation when, for example, you find yourself leaving with some weird result. The state of execution is exactly the same as when you originally called, (unless you use side effects in your program; i.e. asserts etc. will not be undone). When a retry is performed the invocation counter is reset so that counting will continue from the current invocation number regardless of what happened before the retry. This is in accord with the fact that you have, in executional terms, returned to the state before anything else was called.

If you supply an integer argument, then this should denote an invocation number of an ancestral goal. The system tries to get you to the Call port of the box you have specified. It does this by continuously failing until it reaches the right place. Unfortunately this process cannot be guaranteed: it may be the case that the invocation you are looking for has been cut out of the search space by cuts (`!`) in your program. In this case the system fails to the latest surviving Call port before the correct one.

`f` *fail* can be used at any of the four ports (although at the Fail port it has no effect). It transfers control to the Fail port of the box, forcing the invocation to fail prematurely.

If you supply an integer after the command, then this is taken as specifying an invocation number and the system tries to get you to the Fail port of the invocation box you have specified. It does this by continuously failing until it reaches the right place. Unfortunately this process cannot be guaranteed: it may be the case that the invocation you are looking for has been cut out of the search space by cuts (`!`) in your program. In this case the system fails to the latest surviving Fail port before the correct one.

`d` *display goal* displays the current goal using `display/1`. See Write (below).

`p` *print goal* re-prints the current goal. An argument will override the default printdepth, treating 0 as infinity.

`w` *write goal* writes the current goal using `write/1`.

`g` *print ancestor goals* provides you with a list of ancestors to the current goal, i.e. all goals that are hierarchically above the current goal in the calling sequence. You can always be sure of jumping to any goal in the ancestor list (by using retry etc). If you supply an integer *n*, then only that number of ancestors will be printed. That is to say, the last *n* ancestors will be printed counting back from the current goal. Each entry is displayed just as they would be in a trace message.

`t` *print backtrace* is the same as the above, but also shows any goals that have exited non-deterministically and their ancestors. This information shows where there are outstanding choices that the program could backtrack to. If you supply an integer *n*, then only that number of goals will be printed.

Ancestors to the current goal are annotated with the '`Call:`' port, as they have not yet exited, whereas goals that have exited are annotated with the '`Exit:`' port. The backtrace is a tree rather than a stack: to find the parent of a given goal with depth indicator *d*, look for the closest goal above it with depth indicator *d-1*.

`&` *print blocked goals* prints a list of the goals which are currently blocked in the current debugging session together with the variable that each such goal is blocked on (see Section 2.3 [Procedural], page 34). The goals are enumerated from 1 and up. If you supply an integer *n*, then only that goal will be printed. Each entry is preceded by the goal number followed by the variable name.

`n` *nodebug* switches the debugger off. Note that this is the correct way to switch debugging off at a trace point. You cannot use the `@` or `b` options because they always return to the debugger.

`=` *debugging* outputs information concerning the status of the debugging package. See Section 7.14 [Debug Pred], page 120, the built-in `debugging/0`.

`+` *spy this* sets a spy-point on the current goal. With an argument, prompts for a condition to be tested each time the debugger reaches the spy-point. Conditions consist of three parts: a *port*, a *goal* template, and a *test* (arbitrary Prolog goal), usually sharing some variables with the *test* part. For the spy-point to apply, the *port* and *goal* parts must match, and the *test* must succeed, otherwise the spy-point is simply disabled. The following example illustrates how to define a spy-point to only apply at Call ports when a test is satisfied:

```
      4  2  Call: nr([0,1,2,3,4,5,6,7,8,9,...],_267) ? + 1
Goal, Port, Cond: nr(L,_), call, (length(L,N), N<3).
{Spy-point placed on user:nr/2}
 + 4  2  Call: nr([0,1,2,3,4,5,6,7,8,9,...],_267) ? l
 + 32  30  Call: nr([8,9],_2771) ? s
```

`-` *nospy this* removes the spy-point from the current goal, if it exists, and any condition for that spy-point.

`.` *find this* outputs information about where the predicate being called is defined.

`a` *abort* causes an abort of the current execution. All the execution states built so far are destroyed and you are put right back at the top-level. (This is the same as the built-in predicate `abort/0`.)

`b` *break* calls the built-in predicate `break/0`, thus putting you at a recursive top-level with the execution so far sitting underneath you. When you end the break (`^D`) you will be reprompted at the port at which you broke. The new execution is completely separate from the suspended one; the invocation numbers will start again from 1 during the break. The debugger is temporarily switched off as you call the break and will be re-switched on when you finish the break and go back to the old execution. However, any changes to the leashing or to spy-points will remain in effect.

`@` *command* gives you the ability to call arbitrary Prolog goals. It is effectively a one-off *break* (see above). The initial message '`| :- `' will be output on the standard error stream, and a command is then read from the standard input stream and executed as if you were at top level.

`u` *unify* is available at the Call port and gives you the option of providing a solution to the goal from the standard input stream rather than executing the goal. This is convenient e.g. for providing a "stub" for a predicate that has not yet been written. A prompt will be output on the standard error stream, and the solution is then read from the standard input stream and unified with the goal.

`e` *raise exception* is available at all ports. A prompt will be output on the standard error stream, and an exception term is then read from the standard input stream and raised in the program being debugged.

`<` This command, without arguments, resets the printdepth to 10. With an argument of *n*, the printdepth is set to *n*, treating 0 as infinity.

`^` While at a particular port, a current *subterm* of the current goal is maintained. It is the current subterm which is displayed, printed, or written when prompting for a debugger command. Used in combination with the printdepth, this provides a means for navigating in the current goal for focusing on the part which is of interest. The current subterm is set to the current goal when arriving at a new port. This command, without arguments, resets the current subterm to the current goal. With an argument of *n* (`>` 0), the current subterm is replaced by its *n*:th subterm. With an argument of *0*, the current subterm is replaced by its parent term. With a list of arguments, the arguments are applied from left to right.

`?`
`h` *help* displays the table of options given above.

6.6 Consulting during Debugging

It is possible, and sometimes useful, to consult a file whilst in the middle of program execution. Predicates, which have been successfully executed and are subsequently redefined by a consult and are later reactivated by backtracking, will not notice the change of their definitions. In other words, it is as if every predicate, when called, creates a virtual copy of its definition for backtracking purposes.

If SICStus Prolog is run via the Emacs interface, the commands for loading code (such as `C-c C-p`, consulting the current predicate) are not directly available when the system prompts you after

printing out a debugging message. Press `b` followed by `RET` to get a recursive top-level, ready to accept the Emacs commands. Type `^D` to return to the debugging port.

6.7 Catching Exceptions

Usually, exceptions that occur during debugging sessions are displayed only in trace mode and for invocation boxes for predicates with spy-points on them, and not during skips. However, it is sometimes useful to make exceptions trap to the debugger at the earliest opportunity instead. The following predicate provides such a possibility

`error_exception(+`*Exception*`)`
`user:error_exception(+`*Exception*`)`

A hook predicate. This predicate is called at all exception ports. If it succeeds, the debugger enters trace mode and prints an exception port message. Otherwise, the debugger mode is unchanged and a message is printed only in trace mode or if a spy-point is reached, and not during skips.

7 Built-In Predicates

It is not possible to redefine built-in predicates. An attempt to do so will give an error message. See [Pred Summary], page 379.

SICStus Prolog provides a wide range of built-in predicates to perform the following tasks:

Input / Output
- Reading-in Programs
- Term and Goal Expansion
- Input and Output of Terms
- Character I/O
- Stream I/O
- Dec-10 Prolog File I/O

Arithmetic
Comparison of Terms
Control
Error and Exception Handling
Information about the State of the Program
Meta-Logic
Modification of Terms
Modification of the Program
Internal Database
Blackboard Primitives
All Solutions
Coroutining
Debugging
Execution Profiling
Muse Support
Miscellaneous

The following descriptions of the built-in predicates are grouped according to the above categorization of their tasks.

7.1 Input / Output

There are two sets of file manipulation predicates in SICStus Prolog. One set is inherited from DEC-10 Prolog. These predicates always refer to a file by name. The other set of predicates is modeled after Quintus Prolog and refer to files as streams. Streams correspond to the file pointers used at the operating system level.

A stream can be opened and connected to a filename or file descriptor for input or output by calling the predicate `open/3`. `open/3` will return a reference to a stream. The stream may then be passed as an argument to various I/O predicates. The predicate `close/1` is used for closing a stream. The predicate `current_stream/3` is used for retrieving information about a stream, and for finding the currently existing streams.

Prolog streams can be accessed from C functions as well. See Section 8.5 [SICStus Streams], page 144, for details.

The possible formats of a stream are:

`'$stream'(`*X*`)`
: A stream connected to some file. *X* is an integer.

`user_input`
: An alias initially referring to the UNIX `stdin` stream. The alias can be changed with `prolog_flag/3` and accessed by the C variable `SP_stdin`.

`user_output`
: An alias initially referring to the UNIX `stdout` stream. The alias can be changed with `prolog_flag/3` and accessed by the C variable `SP_stdout`.

`user_error`
: An alias initially referring to the UNIX `stderr` stream. The alias can be changed with `prolog_flag/3` and accessed by the C variable `SP_stderr`.

  This stream is used by the Prolog top level and debugger, and for progress and error messages.

The DEC-10 Prolog I/O predicates manipulate streams implicitly, by maintaining the notion of a *current input stream* and a *current output stream*. The current input and output streams are set to the `user_input` and `user_output` initially and for every new break (see Section 1.9 [Nested], page 16). The predicate `see/1` (`tell/1`) can be used for setting the current input (output) stream to newly opened streams for particular files. The predicate `seen/0` (`told/0`) close the current input (output) stream, and resets it to the standard input (output) stream. The predicate `seeing/1` (`telling/1`) is used for retrieving the filename associated with the current input (output) streams.

The possible formats of a filename are:

`user`
: This "filename" stands for the standard input or output stream, depending on context. Terminal output is only guaranteed to be displayed if the output stream is explicitly flushed.

File
: where *File* is any atom other than `user`, denotes a file *File* (with an optional '`.pl`' suffix when consulting or compiling or an optional '`.ql`' suffix in `load/1`) located using services provided by the operating system. On UNIX systems, Filenames beginning with '`/`' are absolute; other filenames are looked up in the current working directory.

Alias`(File)`
: where *File* is an atom, denotes an alias which must be rewritten using the user defined predicate `user:file_search_path/2` until an atomic filename is obtained.

Filenames beginning with '`~`' or '`$`' are treated specially. For example,

'`~/sample.pl`'
: is equivalent to '`/home/sics/al/sample.pl`', if '`/home/sics/al`' is the user's home directory. (This is also equivalent to '`$HOME/sample.pl`' as explained below.)

'`~clyde/sample.pl`'
: is equivalent to '`/home/sics/clyde/sample.pl`', if '`/home/sics/clyde`' is Clyde's home directory.

'`$UTIL/sample.pl`'
: is equivalent to '`/usr/local/src/utilities/sample.pl`', provided the value of the environment variable `UTIL` is '`/usr/local/src/utilities`'.

Failure to open a file normally causes an exception to be raised. This behavior can be turned off and on by of the built-in predicates `nofileerrors/0` and `fileerrors/0` described below.

7.1.1 Reading-in Programs

When the predicates discussed in this section are invoked, filenames are relative to the current working directory. During the load, the current working directory is temporarily changed to the directory containing the file being read in. This has the effect that if one of these predicates is invoked recursively, the filename of the recursive load is relative to the directory of the enclosing load. See Chapter 5 [Load Intro], page 51, for an introduction to these predicates.

Directives will be executed in order of occurrence. Be aware of the changed current working directory as it could have an effect on the semantics of directives. Only the first solution of directives is produced, and variable bindings are not displayed. Directives that fail or raise exceptions cause warnings, but do not terminate the load.

Most of the predicates listed below take an argument *Files* which is a single file name or a list of file names. Source files usually end with an '`.pl`' suffix and object file names with an '`.ql`' suffix. These suffixes are optional. Each file name may optionally be prefixed by a module name. The module name specifies where to import the exported predicates of a module file, or where to store the predicates of a non-module file. The module is created if it doesn't exist already.

`absolute_file_name/2` (see Section 7.1.5 [Stream Pred], page 89) is used to look up the files. The file name `user` is reserved and denotes the standard input stream.

These predicates are available in Runtime Systems with the following limitations:

- The compiler is not available, so compiling is replaced by consulting.
- The `redefine_warnings` and `single_var_warnings` Prolog flags have no effect.
- Progress and warning messages are not issued.
- The user is not prompted in the event of name clashes etc.

`load_files(`*:Files*`)`
`load_files(`*:Files*`, +`*Options*`)`
: A generic predicate for loading files with a list of options to provide extra control. This predicate in fact subsumes the other predicates except `use_module/3` which also returns the name of the loaded module. *Options* is a list of zero or more of the following:

`if(`*X*`)` `true` (the default) to always load, or `changed` to load only if the file has not yet been loaded or if it has been modified since it was last loaded. A non-module file is not considered to have been previously loaded if it was loaded into a different module. The file `user` is never considered to have been previously loaded.

`when(`*When*`)`
`always` (the default) to always load, or `compile_time` to load only if the goal is not in the scope of another `load_files/(1,2)` goal loading object code.

The latter is intended for use when the file only defines predicates that are needed for proper term or goal expansion during compilation of other files.

`load_type(`*LoadType)*
`source` to load source code only, `object` to load object code only, or `latest` (the default) to load source or object code, whichever is newest. If the file is `user`, `source` is forced.

`imports(`*Imports*`)`
`all` (the default) to import all exported predicates if the file is a module file, or a list of predicates to import.

`compilation_mode(`*Mode*`)`
`compile` to translate into compiled code, `consult` to translate into static, interpreted code, or `assert_all` to translate into dynamic, interpreted code.

The default is the compilation mode of any ancestor `load_files/(1,2)` goal, or `compile` otherwise. Note that *Mode* has no effect when an object file is loaded, and that it is recommended to use `assert_all` in conjunction with `load_type(source)`, to ensure that the source file will be loaded even in the presence of an object file.

`consult(`*:Files*`)`
`reconsult(`*:Files*`)`
`[]`
`[`*:File*`|+`*Files*`]`
Consults the source file or list of files specified by *File* and *Files*. Same as `load_files(`*Files*`, [load_type(source),compilation_mode(consult)])`.

`compile(`*:Files*`)`
Compiles the source file or list of files specified by *Files*. The compiled code is placed in-core, i.e. is added incrementally to the Prolog database. Same as `load_files(`*Files*`, [load_type(source),compilation_mode(compile)])`.

`load(`*:Files*`)`
Loads the object file or list of files specified by *Files*. Same as `load_files(`*Files*`, [load_type(object)])`.

`ensure_loaded(`*:Files*`)`
: Compiles or loads the file or files specified by *Files* that have been modified after the file was last loaded, or that have not yet been loaded. The recommended style is to use this predicate for non-module files only, but if any module files are encountered, their public predicates are imported. Same as `load_files(`*Files*`, [if(changed)])`.

`use_module(`*:File*`)`
: Compiles or loads the module file specified by *File* if it has been modified after it was last loaded, or not yet been loaded. Its public predicates are imported. The recommended style is to use this predicate for module files only, but any non-module files encountered are simply compiled or loaded. Same as `load_files(`*File*`, [if(changed)])`.

`use_module(`*:File*`, +`*Imports*`)`
: Loads the module file *File* like `ensure_loaded/1` and imports the predicates in *Imports*. If any of these are not public, a warning is issued. *Imports* may also be set to the atom `all` in which case all public predicates are imported. Same as `load_files(`*File*`, [if(changed),imports(Imports)])`.

`use_module(`*?Module*`, :`*File*`, +`*Imports*`)`
: This is equivalent to `use_module/2` with the addition that *Module* is unified with the loaded module after the loading.

`fcompile(`*:Files*`)`
: Compiles the source file or list of files specified by *Files*. If *Files* are prefixed by a module name, that module name will be used for module name expansion during the compilation (see Section 5.3 [Considerations], page 58). The suffix '`.pl`' is added to the given filenames to yield the real source filenames. The compiled code is placed on the object file or list of files formed by adding the suffix '`.ql`' to the given filenames. (This predicate is not available in Runtime Systems.)

`source_file(`*?File*`)`
: *File* is the absolute name of a source file currently in the system.

`source_file(`*:Head*`,`*?File*`)`
`source_file(-`*Head*`,`*?File*`)`
: *Head* is the most general goal for a predicate loaded from *File*.

`require(`*:PredSpecOrSpecs*`)`
: *PredSpecOrSpecs* is a *predicate spec* or a list or a conjunction of such. The predicate will check if the specified predicates are loaded and if not, will try to load or import them using `use_module/2`. The file containing the predicate definitions will be located in the following way:

  - The directories specified with `user:library_directory/1` are searched for a file '`INDEX.pl`'. This file is taken to contain relations between all exported predicates of the module files in the library directory and its subdirectories. If an '`INDEX.pl`' is not found, `require/1` will try to create one by loading the library package `mkindex` and calling `make_index:make_library_index(`*Directory*`)` (see Chapter 11 [The Prolog Library], page 187).
  - The first index entry for the requested predicate will be used to determine the file to load. An exception is raised if the predicate can't be located.

- Once an '`INDEX.pl`' is read, it is cached internally for use in subsequent calls to `require/1`.
- Not available in Runtime Systems.

7.1.2 Term and Goal Expansion

When a program is being read in, SICStus Prolog provides hooks that enable the terms being read in to be source-to-source transformed before the usual processing of clauses or directives. The hooks consist in user-defined predicates that define the transformations. One transformation is always available, however: *definite clause grammars*, a convenient notation for expressing grammar rules. See [Colmerauer 75] and [Pereira & Warren 80].

Definite clause grammars are an extension of the well-known context-free grammars. A grammar rule in Prolog takes the general form

head `-->` *body*.

meaning "a possible form for *head* is *body*". Both *body* and *head* are sequences of one or more items linked by the standard Prolog conjunction operator '`,`'.

Definite clause grammars extend context-free grammars in the following ways:

1. A non-terminal symbol may be any Prolog term (other than a variable or number).
2. A terminal symbol may be any Prolog term. To distinguish terminals from non-terminals, a sequence of one or more terminal symbols is written within a grammar rule as a Prolog list. An empty sequence is written as the empty list '`[]`'. If the terminal symbols are character codes, such lists can be written (as elsewhere) as strings. An empty sequence is written as the empty list, '`[]`' or '`""`'.
3. Extra conditions, in the form of Prolog procedure calls, may be included in the right-hand side of a grammar rule. Such procedure calls are written enclosed in '`{}`' brackets.
4. The left-hand side of a grammar rule consists of a non-terminal, optionally followed by a sequence of terminals (again written as a Prolog list).
5. Disjunction, if-then, if-then-else, and not-provable may be stated explicitly in the right-hand side of a grammar rule, using the operators '`;`' ('`|`'), '`->`', and '`\+`' as in a Prolog clause.
6. The cut symbol may be included in the right-hand side of a grammar rule, as in a Prolog clause. The cut symbol does not need to be enclosed in '`{}`' brackets.

As an example, here is a simple grammar which parses an arithmetic expression (made up of digits and operators) and computes its value.

```
expr(Z) --> term(X), "+", expr(Y), {Z is X + Y}.
expr(Z) --> term(X), "-", expr(Y), {Z is X - Y}.
expr(X) --> term(X).

term(Z) --> number(X), "*", term(Y), {Z is X * Y}.
term(Z) --> number(X), "/", term(Y), {Z is X / Y}.
term(Z) --> number(Z).

number(C) --> "+", number(C).
number(C) --> "-", number(X), {C is -X}.
number(X) --> [C], {"0"=<C, C=<"9", X is C - "0"}.
```

In the last rule, C is the character code of some digit.

The query

```
| ?- expr(Z, "-2+3*5+1", []).
```

will compute Z=14. The two extra arguments are explained below.

Now, in fact, grammar rules are merely a convenient "syntactic sugar" for ordinary Prolog clauses. Each grammar rule takes an input string, analyses some initial portion, and produces the remaining portion (possibly enlarged) as output for further analysis. The arguments required for the input and output strings are not written explicitly in a grammar rule, but the syntax implicitly defines them. We now show how to translate grammar rules into ordinary clauses by making explicit the extra arguments.

A rule such as

```
p(X) --> q(X).
```

translates into

```
p(X, S0, S) :- q(X, S0, S).
```

If there is more than one non-terminal on the right-hand side, as in

```
p(X, Y) -->
        q(X),
        r(X, Y),
        s(Y).
```

then corresponding input and output arguments are identified, as in

```
p(X, Y, S0, S) :-
        q(X, S0, S1),
        r(X, Y, S1, S2),
        r(Y, S2, S).
```

Terminals are translated using the built-in predicate `'C'(`*S1*`,` *X*`,` *S2*`)`, read as "point *S1* is connected by terminal *X* to point *S2*", and defined by the single clause

```
'C'([X|S], X, S).
```

(This predicate is not normally useful in itself; it has been given the name upper-case `C` simply to avoid using up a more useful name.) Then, for instance

```
p(X) --> [go,to], q(X), [stop].
```

is translated by

```
p(X, S0, S) :-
        'C'(S0, go, S1),
        'C'(S1, to, S2),
        q(X, S2, S3),
        'C'(S3, stop, S).
```

Extra conditions expressed as explicit procedure calls naturally translate as themselves, e.g.

```
p(X) --> [X], {integer(X), X>0}, q(X).
```

translates to

```
p(X, S0, S) :-
        'C'(S0, X, S1),
        integer(X),
        X>0,
        q(X, S1, S).
```

Similarly, a cut is translated literally.

Terminals are translated using the built-in predicate `'C'`(*S1*, *X*, *S2*), read as "point *S1* is connected by terminal *X* to point *S2*", and defined by the single clause

Terminals on the left-hand side of a rule are also translated using `'C'/3`, connecting them to the output argument of the head non-terminal, e.g.

```
is(N), [not] --> [aint].
```

becomes

```
is(N, S0, S) :-
        'C'(S0, aint, S1),
        'C'(S, not, S1).
```

Disjunction has a fairly obvious translation, e.g.

```
args(X, Y) -->
        (   dir(X), [to], indir(Y)
        ;   indir(Y), dir(X)
        ).
```

translates to

```
args(X, Y, S0, S) :-
        (   dir(X, S0, S1),
            'C'(S1, to, S2),
            indir(Y, S2, S)
        ;   indir(Y, S0, S1),
            dir(X, S1, S)
        ).
```

Similarly for if-then, if-then-else, and not-provable.

The built-in predicates which are concerned with grammar rules and other compile/consult time transformations are as follows:

`expand_term(+`*Term1*`,?`*Term2*`)`
: If *Term1* is a term that can be transformed, *Term2* is the result. Otherwise *Term2* is just *Term1* unchanged. This transformation takes place automatically when grammar rules are read in, but sometimes it is useful to be able to perform it explicitly. Grammar rule expansion is not the only transformation available; the user may define clauses for the predicate `user:term_expansion/(2,4)` to perform other transformations. `user:term_expansion(`*Term1*`[,`*Layout1*`],`*Term2*`[,`*Layout2*`])` is called first, and only if it fails is the standard expansion used.

`term_expansion(+`*Term1*`,?`*TermOrTerms*`)`
`user:term_expansion(+`*Term1*`,?`*TermOrTerms*`)`
`term_expansion(+`*Term1*`,+`*Layout1*`,?`*TermOrTerms*`,?`*Layout2*`)`
`user:term_expansion(+`*Term1*`,+`*Layout1*`,?`*TermOrTerms*`,?`*Layout2*`)`
: *A hook predicate*, which defines transformations on terms read while a program is consulted or compiled. It is called for every *Term1* read, including at end of file, represented as the term `end_of_file`. If it succeeds, *TermOrTerms* is used for further processing, otherwise the default grammar rule expansion is attempted. It is often useful to let a term expand to a list of commands and clauses, which will then be processed sequentially.

  The 4 arguments version also defines transformations on the layout of the term read, so that the source-linked debugger can display accurate source code lines if the transformed code needs debugging. *Layout1* is the layout corresponding to *Term1*, and *Layout2* should be a valid layout of *TermOrTerms* (see Section 7.1.3 [Term I/O], page 80).

  For accessing aspects of the load context, e.g. the name of the file being compiled, the predicate `prolog_load_context/2` (see Section 7.6 [State Info], page 104) can be used.

  `user:term_expansion/(2,4)` may also be used to transform queries entered at the terminal in response to the '`| ?- `' prompt. In this case, it will be called with *Term1* = `?-(`*Query*`)` and should succeed with *TermOrTerms* = `?-(`*ExpandedQuery*`)`.

`goal_expansion(+`*Goal*`,+`*Module*`,?`*NewGoal*`)`
`user:goal_expansion(+`*Goal*`,+`*Module*`,?`*NewGoal*`)`
: *A hook predicate.* This predicate defines transformations on goals while clauses are being consulted, compiled or asserted, *after* any processing by `user:term_expansion/(2,4)` of the terms being read in. It is called for every simple *Goal* encountered in the calling context *Module*. If it succeeds, *Goal* is replaced by *NewGoal*,

otherwise *Goal* is left unchanged. *NewGoal* may be an arbitrarily complex goal, and `user:goal_expansion/3` is recursively applied to its subgoals.

This predicate is also used to resolve meta-calls to *Goal* at runtime via the same mechanism. If the transformation succeeds, *NewGoal* is simply called instead of *Goal*. Otherwise, if *Goal* is a goal of an existing predicate, that predicate is invoked. Otherwise, error recovery is attempted by `user:unknown_predicate_handler/3` as described below.

`user:goal_expansion/3` can be regarded as a macro expansion facility. It is used for this purpose to support the interface to attributed variables in `library(atts)`, which defines the predicates *M*`:get_atts/2` and *M*`:put_atts/2` to access module-specific variable attributes. These "predicates" are actually implemented via the `user:goal_expansion/3` mechanism. This has the effect that calls to the interface predicates are expanded at compile time to efficient code.

For accessing aspects of the load context, e.g. the name of the file being compiled, the predicate `prolog_load_context/2` (see Section 7.6 [State Info], page 104) can be used.

`phrase(`*:Phrase*`,`*?List*`)`
`phrase(`*:Phrase*`,`*?List*`,`*+Remainder*`)`
: The list *List* is a phrase of type *Phrase* (according to the current grammar rules), where *Phrase* is either a non-terminal or more generally a grammar rule body. *Remainder* is what remains of the list after a phrase has been found. If called with 2 arguments, the remainder has to be the empty list.

`'C'(`*?S1*`,`*?Terminal*`,`*?S2*`)`
: Not normally of direct use to the user, this built-in predicate is used in the expansion of grammar rules (see above). It is defined as if by the clause `'C'([X|S], X, S).`

7.1.3 Input and Output of Terms

Most of the following predicates come in two versions, with or without a stream argument. Predicates without a stream argument operate on the current input or output stream, depending on context.

Some of these predicates support a notation for terms containing multiple occurrences of the same subterm (cycles and DAGs). The notation is `@(`*Template*`,`*Substitution*`)` where *Substitution* is a list of *Var*=*Term* pairs where the *Var* occurs in *Template* or in one of the *Terms*. This notation stands for the instance of *Template* obtained by binding each *Var* to its corresponding *Term*. The purpose of this notation is to provide a finite printed representation of cyclic terms. This notation is not used by default, and `@/2` has no special meaning except in this context.

`read(`*?Term*`)` *[ISO]*
`read(+`*Stream*`,`*?Term*`)` *[ISO]*
: The next term, delimited by a full-stop (i.e. a `.`, possibly followed by layout text), is read from *Stream* and is unified with *Term*. The syntax of the term must agree with current operator declarations. If a call `read(`*Stream, Term*`)` causes the end of *Stream* to be reached, *Term* is unified with the term `end_of_file`. Further calls to `read/2`

for the same stream will then raise an exception, unless the stream is connected to the terminal.

`read_term(`*?Term*`,+`*Options*`)` *[ISO]*
`read_term(+`*Stream*`,?`*Term*`,+`*Options*`)` *[ISO]*
Same as `read/(1-2)` with a list of options to provide extra control or information about the term. *Options* is a list of zero or more of:

`syntax_errors(+`*Val*`)`
Controls what action to take on syntax errors. *Val* must be one of the values allowed for the `syntax_errors` Prolog flag. The default is set by that flag.

`variable_names(`*?Names*`)`
Names is bound to a list of *Name*=*Var* pairs, where each *Name* is an atom indicating the name of a non-anonymous variable in the term, and *Var* is the corresponding variable.

`singletons(`*?Names*`)`
Names is bound to a list of *Name*=*Var* pairs, one for each variable appearing only once in the term and whose name does not begin with `_`.

`cycles(+`*Boolean*`)`
Boolean must be `true` or `false`. If selected, any occurrences of `@/2` in the term read in are replaced by the potentially cyclic terms they denote as described above. Otherwise (the default), *Term* is just unified with the term read in.

`layout(`*?Layout*`)`
Layout is bound to a *layout term* corresponding to *Term*. The layout Y of a term X is one of:

- If X is a variable or atomic term, Y is the number of the line where X occurs.
- If X is a compound term, Y is a list whose head is the number of the line where the first token of X occurs, and whose remaining elements are the layouts of the arguments of X.
- `[]`, if no line number information is available for X.

```
| ?- read_term(T, [layout(L), variable_names(Va), singletons(S)]).
|: [
     foo(X),
     X = Y
     ].

L = [35,[36,36],[36,[37,37,37],38]],
S = ['Y'=_A],
T = [foo(_B),_B=_A],
Va = ['X'=_B,'Y'=_A]
```

`write(`*?Term*`)` *[ISO]*
`write(+`*Stream*`,`*?Term*`)` *[ISO]*
: The term *Term* is written onto *Stream* according to current operator declarations. Same as `write_term([`*Stream*`,] `*Term*`, [numbervars(true)])`.

`display(`*?Term*`)`
: The term *Term* is displayed *onto the standard output stream* (which is not necessarily the current output stream) in standard parenthesized prefix notation. Same as `write_term(user, `*Term*`, [ignore_ops(true)])`.

`write_canonical(`*?Term*`)` *[ISO]*
`write_canonical(+`*Stream*`,`*?Term*`)` *[ISO]*
: Similar to `write(`*Stream*`,`*Term*`)`. The term will be written according to the standard syntax. The output from `write_canonical/2` can be parsed by `read/2` even if the term contains special characters or if operator declarations have changed. Same as `write_term([`*Stream*`,] `*Term*`, [quoted(true),ignore_ops(true)])`.

`writeq(`*?Term*`)` *[ISO]*
`writeq(+`*Stream*`,`*?Term*`)` *[ISO]*
: Similar to `write(`*Stream*`,`*Term*`)`, but the names of atoms and functors are quoted where necessary to make the result acceptable as input to `read/2`, provided the same operator declarations are in effect. Same as `write_term([`*Stream*`,] `*Term*`, [quoted(true),numbervars(true)])`.

`print(`*?Term*`)`
`print(+`*Stream*`,`*?Term*`)`
: *Hookable.* Prints *Term* onto *Stream*. This predicate provides a handle for user defined pretty printing:

  - If *Term* is a variable then it is output using `write(`*Stream*`,`*Term*`)`.
  - If *Term* is non-variable then a call is made to the user defined predicate `user:portray/1`. If this succeeds then it is assumed that *Term* has been output.
  - Otherwise `print/2` is called recursively on the components of *Term*, unless *Term* is atomic in which case it is written via `write/2`.

  In particular, the debugging package prints the goals in the tracing messages, and the top-level prints the final values of variables. Thus you can vary the forms of these messages if you wish.

  Note that on lists (`[_|_]`), `print/2` will first give the whole list to `user:portray/1`, but if this fails it will only give each of the (top level) elements to `user:portray/1`. That is, `user:portray/1` will not be called on all the tails of the list.

  Same as `write_term([`*Stream*`,] `*Term*`, [portrayed(true),numbervars(true)])`.

`portray(+`*Term*`)`
`user:portray(+`*Term*`)`
: *A hook predicate.* This should either print the *Term* and succeed, or do nothing and fail. In the latter case, the default printer (`write/1`) will print the *Term*.

`portray_clause(+`*Clause*`)`
`portray_clause(+`*Stream*`,+`*Clause*`)`
: Writes the clause *Clause* onto *Stream* exactly as `listing/(0-1)` would have written it. Same as `write_term([`*Stream*`,] `*Term*`, [quoted(true),numbervars(true),indented(true)])` followed by a period and a newline, removing redundant module prefixes and binding variables to terms of the form `'$VAR'(`*N*`)` yielding friendlier variable names.

`write_term(+`*Term*`,+`*Options*`)` *[ISO]*
`write_term(+`*Stream*`,+`*Term*`,+`*Options*`)` *[ISO]*
: Same as `write/(1-2)` etc. with a list of options to provide extra control. This predicate in fact subsumes the above output predicates except `portray_clause/(1,2)` which additionally prints a period and a newline, and removes module prefixes that are redundant wrt. the current type-in module. *Options* is a list of zero or more of the following, where *Boolean* must be `true` or `false` (`false` is the default).

  `quoted(+`*Boolean*`)`
  : If selected, functors are quoted where necessary to make the result acceptable as input to `read/1`. `write_canonical/1`, `writeq/1`, and `portray_clause/1` select this.

  `ignore_ops(+`*Boolean*`)`
  : If selected, *Term* is written in standard parenthesized notation instead of using operators. `write_canonical/1` and `display/1` select this.

  `portrayed(+`*Boolean*`)`
  : If selected, `user:portray/1` is called for each subterm. `print/1` selects this.

  `numbervars(+`*Boolean*`)`
  : If selected, occurrences of `'$VAR'(`*N*`)` where *N* is an integer `>=` 0 are treated specially (see `numbervars/3`). `print/1`, `write/1`, `writeq/1`, and `portray_clause/1` select this.

  `cycles(+`*Boolean*`)`
  : If selected, the potentially cyclic term is printed in finite `@/2` notation, as discussed above.

  `indented(+`*Boolean*`)`
  : If selected, the term is printed with the same indentation as is used by `portray_clause/1` and `listing/(0-1)`.

  `max_depth(`*N*`)`
  : Depth limit on printing. *N* is an integer. 0 (the default) means no limit.

`format(+`*Format*`,+`*Arguments*`)`
`format(+`*Stream*`,+`*Format*`,+`*Arguments*`)`
: Prints *Arguments* onto *Stream* according to format *Format*. *Format* is a list of formatting characters. If *Format* is an atom then `name/2` (see Section 7.7 [Meta Logic], page 110) will be used to translate it into a list of characters. Thus

```
| ?- format("Hello world!", []).
```

  has the same effect as

```
| ?- format('Hello world!', []).
```

`format/2` and `format/3` is a Prolog interface to the C `stdio` function `printf()`. It is modeled after and compatible with Quintus Prolog.

Arguments is a list of items to be printed. If there are no items then an empty list should be supplied.

The default action on a format character is to print it. The character `~` introduces a control sequence. To print a `~` repeat it:

```
| ?- format("Hello ~~world!", []).
```

will result in

```
Hello ~world!
```

The escape sequence (see [Escape Sequences], page 401) `\c` (c for continue) is useful when formatting a string for readability. It causes all characters up to, but not including, the next non-layout character to be ignored.

```
| ?- format("Hello \c
             world!", []).
```

will result in

```
Hello world!
```

The general format of a control sequence is '~*NC*'. The character *C* determines the type of the control sequence. *N* is an optional numeric argument. An alternative form of *N* is '`*`'. '`*`' implies that the next argument in *Arguments* should be used as a numeric argument in the control sequence. Example:

```
| ?- format("Hello~4cworld!", [0'x]).
```

and

```
| ?- format("Hello~*cworld!", [4,0'x]).
```

both produce

```
Helloxxxxworld!
```

The following control sequences are available.

'`~a`' The argument is an atom. The atom is printed without quoting.

'~*N*`c`' (Print character.) The argument is a number that will be interpreted as a character code. *N* defaults to one and is interpreted as the number of times to print the character.

'~*N*`e`'
'~*N*`E`'
'~*N*`f`'
'~*N*`g`'
'~*N*`G`' (Print float). The argument is a float. The float and *N* will be passed to the C `printf()` function as

```
printf("%.Ne", Arg)
printf("%.NE", Arg)
printf("%.Nf", Arg)
printf("%.Ng", Arg)
printf("%.NG", Arg)
```

respectively.

If N is not supplied the action defaults to

```
printf("%e", Arg)
printf("%E", Arg)
printf("%f", Arg)
printf("%g", Arg)
printf("%G", Arg)
```

respectively.

'~*N*d' (Print decimal.) The argument is an integer. N is interpreted as the number of digits after the decimal point. If N is 0 or missing, no decimal point will be printed. Example:

```
| ?- format("Hello ~1d world!", [42]).
Hello 4.2 world!

| ?- format("Hello ~d world!", [42]).
Hello 42 world!
```

'~*N*D' (Print decimal.) The argument is an integer. Identical to '~*N*d' except that ',' will separate groups of three digits to the left of the decimal point. Example:

```
| ?- format("Hello ~1D world!", [12345]).
Hello 1,234.5 world!
```

'~*N*r' (Print radix.) The argument is an integer. N is interpreted as a radix, $2 =< N =< 36$. If N is missing the radix defaults to 8. The letters 'a-z' will denote digits larger than 9. Example:

```
| ?- format("Hello ~2r world!", [15]).
Hello 1111 world!

| ?- format("Hello ~16r world!", [15]).
Hello f world!
```

'~*N*R' (Print radix.) The argument is an integer. Identical to '~*N*r' except that the letters 'A-Z' will denote digits larger than 9. Example:

```
| ?- format("Hello ~16R world!", [15]).
Hello F world!
```

'~*N*s' (Print string.) The argument is a list of character codes. Exactly N characters will be printed. N defaults to the length of the string. Example:

```
| ?- format("Hello ~4s ~4s!", ["new","world"]).
Hello new  worl!

| ?- format("Hello ~s world!", ["new"]).
Hello new world!
```

'~i' (Ignore.) The argument, which may be of any type, is ignored. Example:

```
| ?- format("Hello ~i~s world!", ["old","new"]).
Hello new world!
```

'`~k`' (Print canonical.) The argument may be of any type. The argument will be passed to `write_canonical/1` (see Section 7.1.3 [Term I/O], page 80). Example:

```
| ?- format("Hello ~k world!", [[a,b,c]]).
Hello .(a,.(b,.(c,[]))) world!
```

'`~p`' (Print.) The argument may be of any type. The argument will be passed to `print/1` (see Section 7.1.3 [Term I/O], page 80). Example:

```
| ?- assert((portray([X|Y]) :- print(cons(X,Y)))).
| ?- format("Hello ~p world!", [[a,b,c]]).
Hello cons(a,cons(b,cons(c,[]))) world!
```

'`~q`' (Print quoted.) The argument may be of any type. The argument will be passed to `writeq/1` (see Section 7.1.3 [Term I/O], page 80). Example:

```
| ?- format("Hello ~q world!", [['A','B']]).
Hello ['A','B'] world!
```

'`~w`' (Write.) The argument may be of any type. The argument will be passed to `write/1` (see Section 7.1.3 [Term I/O], page 80). Example:

```
| ?- format("Hello ~w world!", [['A','B']]).
Hello [A,B] world!
```

'`~@`' (Call.) The argument is a goal, which will be called and expected to print on the current output stream. If the goal is not a built-in predicate, it should be module prefixed, as `format/(2-3)` are not meta-predicates. If the goal performs other side-effects or does not succeed deterministically, the behavior is undefined. Example:

```
| ?- format("Hello ~@ world!", [write(new)]).
Hello new world!
```

'`~~`' (Print tilde.) Takes no argument. Prints '`~`'. Example:

```
| ?- format("Hello ~~ world!", []).
Hello ~ world!
```

'`~`*N*`n`' (Print newline.) Takes no argument. Prints *N* newlines. *N* defaults to 1. Example:

```
| ?- format("Hello ~n world!", []).
Hello
 world!
```

'`~N`' (Print Newline.) Prints a newline if not at the beginning of a line.

The following control sequences set column boundaries and specify padding. A column is defined as the available space between two consecutive column boundaries on the same line. A boundary is initially assumed at line position 0. The specifications only apply to the line currently being written.

When a column boundary is set ('`~|`' or '`~+`') and there are fewer characters written in the column than its specified width, the remaining space is divided equally amongst the pad sequences ('`~t`') in the column. If there are no pad sequences, the column is space padded at the end.

If '~|' or '~+' specifies a position preceding the current position, the boundary is set at the current position.

'~*N*|' Set a column boundary at line position *N*. *N* defaults to the current position.

'~*N*+' Set a column boundary at *N* positions past the previous column boundary. *N* defaults to 8.

'~*N*t' Specify padding in a column. *N* is the fill character code. *N* may also be specified as '*C* where *C* is the fill character. The default fill character is SPC. Any ('~t') after the last column boundary on a line is ignored.

Example:

```
| ?-
        format("~`*t NICE TABLE ~`*t~61|~n", []),
        format("*~t*~61|~n", []),
        format("*~t~a~20|~t~a~t~20+~a~t~20+~t*~61|~n",
               ['Right aligned','Centered','Left aligned']),
        format("*~t~d~20|~t~d~t~20+~d~t~20+~t*~61|~n",
               [123,45,678]),
        format("*~t~d~20|~t~d~t~20+~d~t~20+~t*~61|~n",
               [1,2345,6789]),
        format("~`*t~61|~n", []).

********************** NICE TABLE ************************
*                                                        *
*      Right aligned      Centered      Left aligned      *
*                123         45         678              *
*                  1        2345        6789             *
**********************************************************
```

7.1.4 Character Input/Output

There are two sets of character I/O predicates. The first set uses the current input and output streams, while the second set always uses the standard input and output streams. The first set is available in an alternative version where the stream is specified explicitly. The rule is that the stream is the first argument, which defaults to the current input or output stream, depending on context.

nl *[ISO]*
nl(+*Stream*) *[ISO]*

A new line is started on *Stream* by printing an LFD. If *Stream* is connected to the terminal, its buffer is flushed.

get0(*?N*) *[ISO]*
get0(+*Stream*,*?N*) *[ISO]*

N is the character code of the next character read from *Stream*. If all characters of *Stream* have been read, *N* is -1, and further calls to get0/2 for the same stream will then raise an exception, unless the stream is connected to the terminal.

These predicates are called `get_code/(1-2)` in the ISO Prolog standard.

`peek_char(`*?N*`)` *[ISO]*
`peek_char(+`*Stream*`,`*?N*`)` *[ISO]*
: *N* is the character code of the next character from *Stream*, or -1, if all characters of *Stream* have been read, *N*. The character is not actually read, it is only looked at and is still available for subsequent input.

  These predicates are called `peek_code/(1-2)` in the ISO Prolog standard.

`get(`*?N*`)`
`get(+`*Stream*`,`*?N*`)`
: Same as `get0/2`, except *N* is the character code of the next character that is not a *layout-char* (see [Token String], page 398) read from *Stream*.

`skip(+`*N*`)`
`skip(+`*Stream*`,+`*N*`)`
: Skips just past the next character code *N* from *Stream*. *N* may be an arithmetic expression.

`skip_line`
`skip_line(+`*Stream*`)`
: Skips just past the next `LFD` from *Stream*.

`put(+`*N*`)` *[ISO]*
`put(+`*Stream*`,+`*N*`)` *[ISO]*
: Character code *N* is output onto *Stream*. *N* may be an arithmetic expression.

  These predicates are called `put_code/(1-2)` in the ISO Prolog standard.

`tab(+`*N*`)`
`tab(+`*Stream*`,+`*N*`)`
: *N* spaces are output onto *Stream*. *N* may be an arithmetic expression.

The above predicates are the ones which are the most commonly used, as they can refer to any streams. The predicates which follow always refer to the standard input and output streams. They are provided for compatibility DEC-10 character I/O, and are actually redundant and easily recoded in terms of the above predicates.

`ttynl` Same as `nl(user)`.

`ttyflush` Same as `flush_output(user)`.

`ttyget0(`*?N*`)`
: Same as `get0(user,` *N*`)`.

`ttyget(`*?N*`)`
: Same as `get(user,` *N*`)`.

`ttyput(+`*N*`)`
: Same as `put(user,` *N*`)`.

`ttyskip(+`*N*`)`
: Same as `skip(user,` *N*`)`.

`ttytab(+`*N*`)`
: Same as `tab(user,` *N*`)`.

7.1.5 Stream I/O

The following predicates manipulate streams. Character and line counts are maintained per stream. All streams connected to the terminal, however, share the same set of counts. For example, writing to `user_output` will advance the counts for `user_input`, if both are connected to the terminal. Bidirectional streams use the same counters for input and output.

`open(+`*FileName*`,+`*Mode*`,-`*Stream*`)` *[ISO]*
`open(+`*FileName*`,+`*Mode*`,-`*Stream*`,+`*Options*`)` *[ISO]*

If *FileName* is a valid file name, the file is opened in mode *Mode* (invoking the UNIX function `fopen`) and the resulting stream is unified with *Stream*. *Mode* is one of:

`read` Open the file for input.

`write` Open the file for output. The file is created if it does not already exist, the file will otherwise be truncated.

`append` Open the file for output. The file is created if it does not already exist, the file will otherwise be appended to.

If *FileName* is an integer, it is assumed to be a file descriptor passed to Prolog from C. The file descriptor is connected to a Prolog stream (invoking the POSIX function `fdopen`) which is unified with *Stream*.

Options is a list of zero or more of:

`type(+`*T*`)` Specifies whether the stream is a `text` or `binary` stream. Default is `text`.

`close(+`*X*`)` *[ISO]*

If *X* is a stream, the stream is closed. If *X* is the name of a file opened by `see/1` or `tell/1`, the corresponding stream is closed.

`absolute_file_name(+`*RelativeName*`,-`*AbsoluteName*`)`

True if *RelativeName* can be expanded to an absolute file name (an atom) *AbsoluteName*, according to the filename syntax rules (see Section 7.1 [Input Output], page 71). This predicate will first search for a file with the suffix '`.pl`' added to the name given as an argument. If this fails, it will look for a file with no extra suffix added. If a file is found *AbsoluteName* is its absolute file name is returned. Otherwise, *AbsoluteName* is a valid expansion of *RelativeName*,

`absolute_file_name/2` does not produce alternative expansions via backtracking.

If *RelativeName* is `user`, then *AbsoluteName* is also unified with `user`; this "filename" stands for the standard input or output stream, depending on context.

Variants of this predicate are used by all predicates that refer to filenames for resolving these. Predicates that load code require that the specified file exist, possibly with a '`.pl`' or '`.ql`' extension.

`file_search_path(`*Alias*`,?`*Expansion*`)`
`user:file_search_path(`*Alias*`,?`*Expansion*`)`

A hook predicate. This predicate specifies a set of possible file name expansions to be tried when a file specification of the form *Alias*`(`*Name*`)` is used in a predicate that accepts a filename argument.

Alias and *Name* must be atoms. There are two possible cases for *Expansion*.

- *Expansion* is an atom. In this case, the filename expands to *Expansion/Name*, which is subject to further expansion if the resulting atom begins with '`~`' or '`$`' (see Section 7.1 [Input Output], page 71).
- *Expansion* is a compound term *NewAlias*`(NewName)`. In this case, the filename expands to *NewAlias*`(`*NewName*`/`*Name*`)`, which is subject to further expansion via a recursive call to `user:file_search_path/2`.

Since it is possible to have multiple definitions for the same alias, predicates such as `compile/1` may have to explore several alternative expansions before they locate the file to compile. Predicates such as `compile/1` look for a file with a '`.pl`' suffix as well as for a file without the suffix. `load/1` looks for a file with a '`.ql`' suffix.

Aliases are useful in writing portable code, as they minimize the number of places where a program has to be changed if the program or its environment is moved to a different location.

`file_search_path/2` is always called in the `user` module. The predicate exists as a dynamic, multifile predicate at startup with a single clause defining an expansion for the `library/1` alias:

```
file_search_path(library,Path) :- library_directory(Path).
```

`library_directory(`*?Directory*`)`
`user:library_directory(`*?Directory*`)`

A hook predicate. This predicate specifies a set of directories to be searched when a file specification of the form `library(`*Name*`)` is used. The predicate exists as a dynamic, multifile predicate at startup.

Directories to be searched may be added by using `asserta/1` or `assertz/1` (see Section 7.9 [Modify Prog], page 113):

```
| ?- assertz(user:file_search_path(home, '$HOME')).
| ?- assertz(user:file_search_path(demo, home('prolog/demo'))).
| ?- assertz(user:library_directory(home('prolog/lib'))).
```

With these declarations, the file name `demo(mydemo)` would expand to `'$HOME/prolog/demo/mydemo'`, where `'$HOME'` is interpreted as an environment variable (the user's home directory). File names of the form `library(mymodule)` would be looked up in `'$HOME/prolog/lib'` if they cannot be found in the default library directory.

`current_input(`*?Stream*`)` *[ISO]*

Stream is the current input stream. The current input stream is also accessed by the C variable `SP_curin`.

`current_output(`*?Stream*`)` *[ISO]*

Stream is the current output stream. The current output stream is also accessed by the C variable `SP_curout`.

`current_stream(`*?FileName*`,`*?Mode*`,`*?Stream*`)`

Stream is a stream which was opened in mode *Mode* and which is connected to the absolute file name *Filename* (an atom) or to the file descriptor *Filename* (an integer). This predicate can be used for enumerating all currently open streams through backtracking.

`set_input(+`*Stream*`)` *[ISO]*
: Sets the current input stream to *Stream*.

`set_output(+`*Stream*`)` *[ISO]*
: Sets the current output stream to *Stream*.

`flush_output` *[ISO]*
`flush_output(+`*Stream*`)` *[ISO]*
: Flushes all internally buffered characters for *Stream* to the operating system.

`open_null_stream(-`*Stream*`)`
: Opens an output stream. Everything written to this stream will be thrown away.

`character_count(+`*Stream*`,?`*N*`)`
: *N* is the number of characters read/written on stream *Stream*.

`line_count(+`*Stream*`,?`*N*`)`
: *N* is the number of lines read/written on stream *Stream*.

`line_position(+`*Stream*`,?`*N*`)`
: *N* is the number of characters read/written on the current line of *Stream*.

`stream_position(+`*Stream*`,?`*Position*`)`
: *Position* is a term representing the current position of *Stream*. The relative order of stream position terms can be tested with standard term comparison predicates such as `compare/3`, but you should not otherwise rely on their internal representation. This operation is available for any Prolog stream.

`set_stream_position(+`*Stream*`,+`*Position*`)` *[ISO]*
: *Position* is a term representing a new position of *Stream*, which is then set to the new position. This operation is only available for Prolog streams connected to "seekable devices" (disk files, usually).

`seek(+`*Stream*`,+`*Offset*`,+`*Method*`,-`*NewLocation*`)`
: True if the stream *Stream* can be set to the byte offset *Offset* relative to *Method*, and *NewLocation* is the new byte offset from the beginning of the file after the operation. *Method* must be one of:

`bof`	Seek from the beginning of the file stream.
`current`	Seek from the current position of the file stream.
`eof`	Seek from the end of the file stream.

  This operation is only available for Prolog streams connected to "seekable devices" (disk files, usually) and is an interface to the `stdio` functions `fseek` and `ftell`.

`at_end_of_stream` *[ISO]*
`at_end_of_stream(+`*Stream*`)` *[ISO]*
: The end of stream has been reached for *Stream*. An input stream reaches end of stream when all characters (except `EOF`) of the stream have been read. These predicates peek ahead for next input character if there is no character available on the buffer of *Stream*. Unless the stream is to be treated as connected to the terminal (see `SP_force_interactive`, Section 8.8.1 [Initializing the Prolog Engine], page 151), a stream remains at end of stream after `EOF` has been read, and any further attempt to read from the stream will raise an existence error (see Section 7.5 [Exception], page 101).

`at_end_of_line`
`at_end_of_line(+`*Stream*`)`
: The end of stream or end of line has been reached for *Stream*. An input stream reaches end of line when all the characters except `LFD` of the current line have been read. These predicates peek ahead for next input character if there is no character available on the buffer of *Stream*.

`fileerrors`
: Undoes the effect of `nofileerrors/0`.

`nofileerrors`
: After a call to this predicate, failure to locate or open a file will cause the operation to fail instead of the default action, which is to raise an exception with an error message.

`stream_select(+`*Streams*`,+`*TimeOut*`,-`*ReadStreams*`)`
: The list of streams in *Streams* is checked for readable characters. A stream can be any stream associated with an I/O descriptor. The list *ReadStreams* returns the streams with readable data. If *TimeOut* is instantiated to `off`, the predicate waits until something is available. If *TimeOut* is `S:U` the predicate waits at most `S` seconds and `U` microseconds. Both `S` and `U` must be integers `>=0`. If there is a timeout, *ReadStreams* is `[]`.

  Not available in Muse. Not available in operating systems that do not support the `system()` system call.

`stream_interrupt(?`*Stream*`,?`*OldHandler*`,?`*NewHandler*`)`
: Installs *NewHandler* as an interrupt-handler which is invoked when something is readable on *Stream*. *OldHandler* is the current interrupt handler *Stream* must be associated with an I/O descriptor. Interrupt handlers are specified as atoms. The atom `off` indicates that the interrupt mechanism is turned off for *Stream*. Any other atom is the name of a predicate invoked when something is readable on *Stream*. The handler predicate has one argument, the stream that is readable. For example,

  ```
  stream_interrupt(Stream, _, int_handler).
  ```

  will enable the interrupt mechanism. Given the predicate

  ```
  int_handler(Stream) :-
          read(Stream, Data),
          write(Data), nl.
  ```

  the term read from *Stream* will be written to the current output. **Note:** there is no guarantee that a complete Prolog term is available yet. If not, `read/2` will suspend as usual.

  Not available in Muse. Not available in operating systems that do not provide the ability to generate signals when new data becomes available on a file descriptor.

7.1.6 DEC-10 Prolog File I/O

The following predicates manipulate files.

`see(+`*File*`)`
: The file *File* becomes the current input stream. *File* may be a stream previously opened by `see/1` or a filename. If it is a filename, the following action is taken: If there is a stream opened by `see/1` associated with the same file already, then it becomes the current input stream. Otherwise, the file *File* is opened for input and made the current input stream.

`seeing(?`*FileName*`)`
: *FileName* is unified with the name of the current input file, if it was opened by `see/1`, with the current input stream, if it is not `user_input`, otherwise with `user`.

`seen`
: Closes the current input stream, and resets it to `user_input`.

`tell(+`*File*`)`
: The file *File* becomes the current output stream. *File* may be a stream previously opened by `tell/1` or a filename. If it is a filename, the following action is taken: If there is a stream opened by `tell/1` associated with the same file already, then it becomes the current output stream. Otherwise, the file *File* is opened for output and made the current output stream.

`telling(?`*FileName*`)`
: *FileName* is unified with the name of the current output file, if it was opened by `tell/1`, with the current output stream, if it is not `user_output`, otherwise with `user`.

`told`
: Closes the current output stream, and resets it to `user_output`.

7.1.7 An Example

Here is an example of a common form of file processing:

```
process_file(F) :-
        seeing(OldInput),
        see(F),                 % Open file F
        repeat,
          read(T),              % Read a term
          process_term(T),      % Process it
          T == end_of_file,     % Loop back if not at end of file
        !,
        seen,                   % Close the file
        see(OldInput).
```

The above is an example of a *repeat loop*. Nearly all sensible uses of `repeat/0` follow the above pattern. Note the use of a cut to terminate the loop.

7.2 Arithmetic

Arithmetic is performed by built-in predicates which take as arguments *arithmetic expressions* and evaluate them. An arithmetic expression is a term built from numbers, variables, and functors that represent arithmetic functions. At the time of evaluation, each variable in an arithmetic expression must be bound to a non-variable expression. An expression evaluates to a number, which may be an *integer* or a *float*.

The range of integers is `[-2^2147483616, 2^2147483616)`. Thus for all practical purposes, the range of integers can be considered infinite.

The range of floats is the one provided by the C `double` type, typically `[4.9e-324, 1.8e+308]` (plus or minus).

Only certain functors are permitted in an arithmetic expression. These are listed below, together with an indication of the functions they represent. X and Y are assumed to be arithmetic expressions. Unless stated otherwise, the arguments of an expression may be any numbers and its value is a float if any of its arguments is a float, otherwise the value is an integer. Any implicit coercions are performed with the `integer/1` and `float/1` functions.

`+(`X`)` The value is X.

`-`X The value is the negative of X.

X`+`Y The value is the sum of X and Y.

X`-`Y The value is the difference of X and Y.

X`*`Y The value is the product of X and Y.

X`/`Y The value is the *float* quotient of X and Y.

X`//`Y The value is the *integer* quotient of X and Y.

X `mod` Y The value is the *integer* remainder after dividing X by Y, i.e. `integer(`X`)-integer(`Y`)*(`X`//`Y`)`.

`integer(`X`)`
The value is the closest integer between X and 0, if X is a float, otherwise to X itself.

`float(`X`)` The value is the float equivalent of X, if X is an integer, otherwise to X itself.

X`/\`Y The value is the bitwise conjunction of the integers X and Y.

X`\/`Y The value is the bitwise disjunction of the integers X and Y.

X`#`Y The value is the bitwise exclusive or of the integers X and Y.

`\(`X`)` The value is the bitwise negation of the integer X.

X`<<`Y The value is the integer X shifted left by Y places.

`X>>Y` The value is the integer X shifted right by Y places.

`[`X`]` A list of just one number X evaluates to X. Since a quoted string is just a list of integers, this allows a quoted character to be used in place of its character code; e.g. `"A"` behaves within arithmetic expressions as the integer 65.

SICStus Prolog also includes an extra set of functions listed below. These may not be supported by other Prologs. All trigonometric and transcendental functions take float arguments and deliver float values. The trigonometric functions take arguments or deliver values in radians.

`abs(`*X*`)` The value is the absolute value of *X*.

`gcd(`*X*`,`*Y*`)`
The value is the greatest common divisor of the two integers *X* and *Y*.

`min(`*X*`,`*Y*`)`
The value is the lesser value of *X* and *Y*.

`max(`*X*`,`*Y*`)`
The value is the greater value of *X* and *Y*.

`msb(`*X*`)` The value is the most significant bit position of the integer *X*. It is equivalent to, but more efficient than, `integer(log(2,X))`.

`round(`*X*`)` The value is the float that is the closest integral value to *X*. If *X* is exactly half-way between two integers, it must be rounded to the closest even integral value.

`truncate(`*X*`)`
The value is the float that is the closest integer between *X* and 0.

`floor(`*X*`)` The value is the float that is the greatest integral value less or equal to *X*.

`ceiling(`*X*`)`
The value is the float that is the least integral value greater or equal to *X*.

`sin(`*X*`)` The value is the sine of *X*.

`cos(`*X*`)` The value is the cosine of *X*.

`tan(`*X*`)` The value is the tangent of *X*.

`cot(`*X*`)` The value is the cotangent of *X*.

`sinh(`*X*`)` The value is the hyperbolic sine of *X*.

`cosh(`*X*`)` The value is the hyperbolic cosine of *X*.

`tanh(`*X*`)` The value is the hyperbolic tangent of *X*.

`coth(`*X*`)` The value is the hyperbolic cotangent of *X*.

`asin(`*X*`)` The value is the arc sine of *X*.

`acos(`*X*`)` The value is the arc cosine of *X*.

`atan(`*X*`)` The value is the arc tangent of *X*.

`atan2(`*X*`,`*Y*`)`
The value is the four-quadrant arc tangent of *X* and *Y*.

`acot(`*X*`)` The value is the arc cotangent of *X*.

`acot2(`*X*`,`*Y*`)`
The value is the four-quadrant arc cotangent of *X* and *Y*.

`asinh(`*X*`)` The value is the hyperbolic arc sine of *X*.

`acosh(`*X*`)` The value is the hyperbolic arc cosine of *X*.

`atanh(`*X*`)` The value is the hyperbolic arc tangent of *X*.

`acoth(`*X*`)` The value is the hyperbolic arc cotangent of *X*.

`sqrt(`*X*`)` The value is the square root of *X*.

`log(`*X*`)` The value is the natural logarithm of *X*.

`log(`*Base*`,`*X*`)`
The value is the logarithm of *X* in the base *Base*.

`exp(`*X*`)` The value is the natural exponent of *X*.

`exp(`*X*`,`*Y*`)`
The value is *X* raised to the power of *Y*.

`inf` The value is infinity as defined in the IEEE standard.

`nan` The value is not-a-number as defined in the IEEE standard.

Variables in an arithmetic expression which is to be evaluated may be bound to other arithmetic expressions rather than just numbers, e.g.

```
evaluate(Expression, Answer) :- Answer is Expression.

| ?- evaluate(24*9, Ans).
Ans = 216 ?

yes
```

Arithmetic expressions, as described above, are just data structures. If you want one evaluated you must pass it as an argument to one of the built-in predicates listed below. Note that it only evaluates one of its arguments, whereas all the comparison predicates evaluate both of theirs. In the following, *X* and *Y* stand for arithmetic expressions, and *Z* for some term.

Z `is` *X* *[ISO]*
X, which must be an arithmetic expression, is evaluated and the result is unified with *Z*.

X `=:=` *Y* *[ISO]*
The numeric values of *X* and *Y* are equal.

X `=\=` *Y* *[ISO]*
The numeric values of *X* and *Y* are not equal.

X `<` *Y* *[ISO]*
The numeric value of *X* is less than the numeric value of *Y*.

X `>` *Y* *[ISO]*
The numeric value of *X* is greater than the numeric value of *Y*.

X `=<` *Y* *[ISO]*
The numeric value of *X* is less than or equal to the numeric value of *Y*.

X `>=` *Y [ISO]*
: The numeric value of *X* is greater than or equal to the numeric value of *Y*.

7.3 Comparison of Terms

These built-in predicates are meta-logical. They treat uninstantiated variables as objects with values which may be compared, and they never instantiate those variables. They should *not* be used when what you really want is arithmetic comparison (see Section 7.2 [Arithmetic], page 94) or unification.

The predicates make reference to a *standard total ordering* of terms, which is as follows:

- Variables, by age (oldest first—the order is *not* related to the names of variables).
- Floats, in numeric order (e.g. -1.0 is put before 1.0).
- Integers, in numeric order (e.g. -1 is put before 1).
- Atoms, in alphabetical (i.e. character code) order.
- Compound terms, ordered first by arity, then by the name of the principal functor, then by age for mutables and by the arguments in left-to-right order for other terms. Recall that lists are equivalent to compound terms with principal functor `./2`.

For example, here is a list of terms in standard order:

```
[ X, -1.0, -9, 1, fie, foe, X = Y, foe(0,2), fie(1,1,1) ]
```

Note: the standard order is only well-defined for finite (acyclic) terms. There are infinite (cyclic) terms for which no order relation holds. Furthermore, blocking goals (see Section 2.3 [Procedural], page 34) on variables or modifying their attributes (see Chapter 14 [Attributes], page 193) does not preserve their order.

These are the basic predicates for comparison of arbitrary terms:

Term1 `==` *Term2 [ISO]*
: The terms currently instantiating *Term1* and *Term2* are literally identical (in particular, variables in equivalent positions in the two terms must be identical). For example, the query

```
| ?- X == Y.
```

fails (answers '`no`') because *X* and *Y* are distinct uninstantiated variables. However, the query

```
| ?- X = Y, X == Y.
```

succeeds because the first goal unifies the two variables (see Section 7.17 [Misc Pred], page 124).

Term1 `\==` *Term2 [ISO]*
: The terms currently instantiating *Term1* and *Term2* are not literally identical.

Term1 `@<` *Term2 [ISO]*
: The term *Term1* is before the term *Term2* in the standard order.

Term1 `@>` *Term2 [ISO]*
: The term *Term1* is after the term *Term2* in the standard order.

Term1 `@=<` *Term2 [ISO]*
: The term *Term1* is not after the term *Term2* in the standard order.

Term1 `@>=` *Term2 [ISO]*
: The term *Term1* is not before the term *Term2* in the standard order.

Some further predicates involving comparison of terms are:

`?=(`*?X*`,`*?Y*`)`
: *X* and *Y* are either syntactically identical or syntactically non-unifiable.

`compare(`*?Op*`,`*?Term1*`,`*?Term2*`)`
: The result of comparing terms *Term1* and *Term2* is *Op*, where the possible values for *Op* are:

  `=` if *Term1* is identical to *Term2*,

  `<` if *Term1* is before *Term2* in the standard order,

  `>` if *Term1* is after *Term2* in the standard order.

  Thus `compare(=,Term1,Term2)` is equivalent to `Term1 == Term2`.

`sort(+`*List1*`,`*?List2*`)`
: The elements of the list *List1* are sorted into the standard order (see Section 7.3 [Term Compare], page 97) and any identical elements are merged, yielding the list *List2*. (The time and space complexity of this operation is at worst *O(N lg N)* where *N* is the length of *List1*.)

`keysort(+`*List1*`,`*?List2*`)`
: The list *List1* must consist of items of the form *Key-Value*. These items are sorted into order according to the value of *Key*, yielding the list *List2*. No merging takes place. This predicate is *stable*, i.e. if `K-A` occurs before `K-B` in the input, then `K-A` will occur before `K-B` in the output. (The time and space complexity of this operation is at worst *O(N lg N)* where *N* is the length of *List1*.)

7.4 Control

`+`*P* `,` `+`*Q* *[ISO]*
: *P* and *Q*.

`+`*P* `;` `+`*Q* *[ISO]*
: *P* or *Q*.

`!` *[ISO]*
: See Section 2.5 [Cut], page 36.

`\+` `+`*P* *[ISO]*
: Fails if the goal *P* has a solution, and succeeds otherwise. This is not real negation ("*P* is false"), but a kind of pseudo-negation meaning "*P* is not provable". It is defined as if by

```
\+(P) :- P, !, fail.
\+(_).
```

No cuts are allowed in *P*.

Remember that with prefix operators such as this one it is necessary to be careful about spaces if the argument starts with a `(`. For example:

```
| ?- \+ (P,Q).
```

is this operator applied to the conjunction of *P* and *Q*, but

```
| ?- \+(P,Q).
```

would require a predicate `\+ /2` for its solution. The prefix operator can however be written as a functor of one argument; thus

```
| ?- \+((P,Q)).
```

is also correct.

`+`*P* `->` `+`*Q* `;` `+`*R* *[ISO]*
: Analogous to

if *P* then *Q* else *R*

and defined as if by

```
(P -> Q; R) :- P, !, Q.
(P -> Q; R) :- R.
```

except the scope of any cut in *Q* or *R* extends beyond the if-then-else construct. **No cuts are allowed in *P*.**

Note that this form of if-then-else only explores *the first* solution to the goal *P*.

Note also that the `;` is not read as a disjunction operator in this case; instead, it is part of the if-then-else construction.

The precedence of `->` is less than that of `;` (see Section 2.6 [Operators], page 37), so the expression is read as

```
;(->(P,Q),R)
```

`+`*P* `->` `+`*Q* *[ISO]*
: When occurring as a goal, this construction is read as equivalent to

```
(P -> Q; fail)
```

`if(+`*P*`,+`*Q*`,+`*R*`)`

Analogous to

if *P* then *Q* else *R*

but differs from `P -> Q ; R` in that `if(P, Q, R)` explores *all* solutions to the goal *P*. There is a small time penalty for this—if *P* is known to have only one solution of interest, the form `P -> Q ; R` should be preferred.

No cuts are allowed in *P*.

`otherwise`
`true` *[ISO]*

These always succeed. Use of `otherwise/0` is discouraged, because it is not as portable as `true/0`, and because the former may suggest a completely different semantics than the latter.

`false`
`fail` *[ISO]*

These always fail. Use of `false/0` is discouraged, because it is not as portable as `fail/0`, and because the latter has a more procedural flavor to it.

`repeat` *[ISO]*

Generates an infinite sequence of backtracking choices. In sensible code, `repeat/0` is hardly ever used except in *repeat loops*. A repeat loop has the structure

```
Head :-
        ...
        save(OldState),
        repeat,
          generate(Datum),
          action(Datum),
          test(Datum),
        !,
        restore(OldState),
        ...
```

The purpose is to repeatedly perform some *action* on elements which are somehow *generate*d, e.g. by reading them from a stream, until some *test* becomes true. Usually, *generate*, *action*, and *test* are all determinate. Repeat loops cannot contribute to the logic of the program. They are only meaningful if the *action* involves side-effects.

The only reason for using repeat loops instead of a more natural tail-recursive formulation is efficiency: when the *test* fails back, the Prolog engine immediately reclaims any working storage consumed since the call to `repeat/0`.

`call(`*:Term*`)` *[ISO]*
`incore(`*:Term*`)`

:Term If *Term* is instantiated to a term which would be acceptable as the body of a clause, then the goal `call(`*Term*`)` is executed exactly as if that term appeared textually in its place, except that any cut (`!`) occurring in *Term* only cuts alternatives in the execution of *Term*. Use of `incore/1` is not recommended.

If *Term* is not instantiated as described above, an error message is printed and the call fails.

`call_cleanup(`*:Goal*`,`*:Cleanup*`)`

This construction can be used to ensure that *Cleanup* is executed as soon as *Goal* has completed execution, no matter how it finishes. In more detail:

When `call_cleanup/2` with a continuation *C* is called or backtracked into, first *Goal* is called or backtracked into. Then there are four possibilities:

1. *Goal* succeeds deterministically, possibly leaving some blocked subgoals. *Cleanup* is executed with continuation *C*.
2. *Goal* succeeds with some alternatives outstanding. Execution proceeds to *C*. If a cut that removes the outstanding alternatives is encountered, *Cleanup* is executed with continuation to proceed after the cut. Also, if an exception *E* that will be caught by an ancestor of the `call_cleanup/2` *Goal* is raised, *Cleanup* is executed with continuation `raise_exception(`*E*`)`.
3. *Goal* fails. *Cleanup* is executed with continuation `fail`.
4. *Goal* raises an exception *E*. *Cleanup* is executed with continuation `raise_exception(`*E*`)`.

In a typical use of `call_cleanup/2`, *Cleanup* succeeds deterministically after performing some side-effect; otherwise, unexpected behavior may result.

Note that the Prolog top level operates as a read-execute-fail loop, which backtracks into or cuts the query when the user types `;` or `RET` respectively. Also, the predicates `halt/0`, `abort/0`, and `reinitialise/0` are implemented in terms of exceptions. All of these circumstances can trigger the execution of *Cleanup*.

7.5 Error and Exception Handling

The two built-in predicates `on_exception/3` and `raise_exception/1` are used to alter the control flow to meet exception and error conditions. The equivalent of a `raise_exception/1` is also executed by the built-in predicates when errors occur.

`on_exception(`*?Pattern*`,`*:ProtectedGoal*`,`*:Handler*`)` *[ISO]*
`raise_exception(+`*Exception*`)` *[ISO]*

`on_exception/3` calls *ProtectedGoal*. If this succeeds or fails, so does the call to `on_exception/3`. If however, during the execution of *ProtectedGoal*, there is a call to `raise_exception(`*Exception*`)`, then *Exception* is copied and the stack is unwound back to the call to `on_exception/3`, whereupon the copy of *Exception* is unified with *Pattern*. If this unification succeeds, then `on_exception/3` calls the goal *Handler* in order to determine the success or failure of `on_exception/3`. Otherwise, the stack keeps unwinding, looking for an earlier invocation of `on_exception/3`. *Exception* may be any term.

Certain built-in and library predicates rely on the exception mechanism, so it is usually a bad idea to let *Pattern* be a variable, matching any exception. If it must be a variable, the *Handler* should examime the exception and pass it on if it is not relevant to the current invocation.

Instead of the above two predicates, the ISO Prolog standard prescribes the following two. They are functionality equivalent to the above two predicates, but beware of the different argument order:

```
catch(Goal, Catcher, Recovery) :-
        on_exception(Catcher, Goal, Recovery).

throw(Ball) :-
        raise_exception(Ball).
```

In a Development System, any previously uncaught exception is caught and an appropriate error message is printed before returning to the top level. In recursive calls to Prolog from C, uncaught exceptions are returned back to C instead. The printing of these and most other messages in a Development System is handled by the predicate `print_message/2`. The behavior of this predicate can be overridden by defining `user:portray_message/2`, so as to suppress or alter the format of certain messages. These predicates work as follows:

`print_message(+`*Severity*`, +`*Message*`)`

Hookable. Most messages from the system are printed by calling this predicate. Before anything is printed, however, `print_message/2` calls `user:portray_message/2` with the same arguments, so as to give the user a means of intercepting the message before it is actually printed. If `user:portray_message/2` succeeds, nothing is printed, otherwise *Message* is formatted and printed using the default method.

Message is a term that encodes the message to be printed. The format of message terms is subject to change, but can be inspected in the file '`Bips/msgs.pl`' of the SICStus Prolog distribution. *Severity* is a term denoting the severity of the message, and is one of:

`force(`*Severity*`)`
: *Message* should be printed without calling the `user:portray_message/2` hook. This is useful if `user:portray_message/2` has intercepted the message, and now wants to print a reformatted version of it using `print_message/2`.

`error`
: *Message* is an uncaught exception. The execution will normally be aborted and return to the top level. Syntax errors and exceptions that occur while loading files do not necessarily abort the execution, however.

`warning`
: *Message* is a warning (e.g. singleton variables).

`informational`
: *Message* provides information e.g. about files being loaded.

`help`
: *Message* is normally a response to a query.

`portray_message(+`*Severity*`, +`*Message*`)`
`user:portray_message(+`*Severity*`, +`*Message*`)`

A hook predicate. Called by `print_message/2` before printing the message. If this succeeds, the default message for printing *Message* is overridden, and nothing more is printed.

The built-in predicates may raise exceptions as follows on errors. Usually the goal and the argument number causing the error are contained in the exception.

`instantiation_error(`*Goal*`,`*ArgNo*`)`
: *Goal* was called with insufficiently instantiated variables.

`type_error(`*Goal*`,`*ArgNo*`,`*TypeName*`,`*Culprit*`)`
: *Goal* was called with the wrong type of argument(s). *TypeName* is the expected type and *Culprit* what was actually found.

`domain_error(`*Goal*`,`*ArgNo*`,`*Domain*`,`*Culprit*`)`
: *Goal* was called with argument(s) of the right type but with illegal value(s). *Domain* is the expected domain and *Culprit* what was actually found.

`existence_error(`*Goal*`,`*ArgNo*`,`*ObjectType*`,`*Culprit*`,`*Reserved*`)`
: Something does not exist as indicated by the arguments. If the unknown-flag (see `prolog_flag/3`) is set to `error`, this error is raised with *ArgNo* set to 0 when an undefined predicate is called.

`permission_error(`*Goal*`,`*Operation*`,`*ObjectType*`,`*Culprit*`,`*Reserved*`)`
: The *Operation* is not permitted on *Culprit* of the *ObjectType*.

`context_error(`*Goal*`,`*ContextType*`,`*CommandType*`)`
: The *CommandType* is not permitted in *ContextType*.

`syntax_error(`*Goal*`,`*Position*`,`*Message*`,`*Tokens*`,`*AfterError*`)`
: A syntax error was found when reading a term with `read/(1-2)`. This error is raised only if the `syntax_errors` flag (see `prolog_flag/3`) is set to `error`.

`representation_error(`*Goal*`,`*ArgNo*`,`*ErrorType*`)`
: A representation error occurs when the program tries to compute some well-defined value which cannot be represented, such as a compound term with arity > 255.

`consistency_error(`*Goal*`,`*Culprit1*`,`*Culprit2*`,`*Message*`)`
: A consistency error occurs when two otherwise valid values or operations have been specified which are inconsistent with each other.

`system_error(`*Message*`)`
: An error occurred while dealing with the operating system.

It is possible to handle a particular kind of existence errors locally: calls to undefined predicates. This can be done by defining clauses for:

`unknown_predicate_handler(+`*Goal*`,+`*Module*`,-`*NewGoal*`)`
`user:unknown_predicate_handler(+`*Goal*`,+`*Module*`,-`*NewGoal*`)`
: *A hook predicate.* This predicate is called as a result of a call to an undefined predicate. *Goal* is bound to the goal of the undefined predicate and *Module* to the module where the call was made. If this predicate succeeds, *Module:NewGoal* is called; otherwise, the action taken is governed by the `unknown` Prolog flag.

The following example shows an auto-loader for library packages:

```
user:unknown_predicate_handler(Goal, Module, Goal) :-
        functor(Goal, Name, Arity),
        require(Module:(Name/Arity)).
```

7.6 Information about the State of the Program

`listing` Lists onto the current output stream all the clauses in the current interpreted program (in the type-in module; see Section 4.2 [Module Spec], page 48). Clauses listed onto a file can be consulted back.

`listing(`*:Spec*`)`
Lists all interpreted predicates covered by *Spec*, which has the same for as for `spy/1` (see Section 6.3 [Spy-Point], page 62). For example:

```
| ?- listing([concatenate/3, reverse, m:go/(2-3), bar:_]).
```

`current_atom(`*?Atom*`)`
Atom is an atom known to SICStus Prolog. Can be used to enumerate (through backtracking) all currently known atoms, and return each one as *Atom*.

`current_predicate(`*?Name*`,`*:Head*`)` *[ISO]*
`current_predicate(`*?Name*`,`*-Head*`)` *[ISO]*
Name is the name of a user defined or library predicate, and *Head* is the most general goal for that predicate, possibly prefixed by a module name. This predicate can be used to enumerate all user defined or library predicates through backtracking.

`predicate_property(`*:Head*`,`*?Property*`)`
`predicate_property(`*-Head*`,`*?Property*`)`
Head is the most general goal for an existing predicate, possibly prefixed by a module name, and *Property* is a property of that predicate, where the possible properties are

- one of the atoms `built_in` (for built-in predicates) or `compiled` or `interpreted` (for user defined predicates) or `fd_constraint` for FD predicates see Section 30.9 [Defining Primitive Constraints], page 290.
- the atom `dynamic` for predicates that have been declared dynamic (see Section 5.2.2 [Dynamic Declarations], page 54),
- the atom `multifile` for predicates that have been declared multifile (see Section 5.2.1 [Multifile Declarations], page 54),
- the atom `volatile` for predicates that have been declared volatile (see Section 5.2.3 [Volatile Declarations], page 55),
- one or more terms `(block` *Term*`)` for predicates that have block declarations (see Section 5.2.4 [Block Declarations], page 55),
- the atom `exported` or terms `imported_from(`*ModuleFrom*`)` for predicates exported or imported from modules (see Chapter 4 [Module Intro], page 47),
- the term `(meta_predicate` *Term*`)` for predicates that have meta-predicate declarations (see Section 4.6 [Meta Decl], page 50).
- in Muse only, the atom `sync` for predicates that perform side effects that must be synchronized to preserve standard semantics.

- in Muse only, the atom `parallel` for predicates that may be executed in parallel.

This predicate can be used to enumerate all existing predicates and their properties through backtracking.

`current_module(`*?Module*`)`
: *Module* is a module in the system. It can be used to backtrack through all modules present in the system.

`current_module(`*?Module*`, `*?File*`)`
: *Module* is the module defined in *File*.

`module(+`*Module*`)`
: The type-in module is set to *Module*.

`prolog_flag(+`*FlagName*`,`*?OldValue*`,`*?NewValue*`)`
: *OldValue* is the value of the Prolog flag *FlagName*, and the new value of *FlagName* is set to *NewValue*. The possible Prolog flag names and values are:

  `agc_margin`
  : An integer *Margin*. The atoms will be garbage collected when *Margin* new atoms have been created since the last atom garbage collection. Initially 10000.

  `argv`
  : A read-only flag. The value is a list of atoms of the program arguments supplied when the current SICStus Prolog process was started. For example, if SICStus Prolog were invoked with:

    ```
    % sicstus -a hello world 2001
    ```

    then the value will be `[hello,world,'2001']`.

  `compiling`
  : Governs the mode in which `compile/1` and `fcompile/1` operate (see Chapter 5 [Load Intro], page 51).

    `compactcode`
    : Compilation produces byte-coded abstract instructions (the default).

    `fastcode`
    : Compilation produces native machine instructions. Currently only available for 680x0, Sparc, and MIPS platforms.

    `profiledcode`
    : Compilation produces byte-coded abstract instructions instrumented to produce execution profiling data.

    `debugcode`
    : Compiling is replaced by consulting.

  `debugging`
  : Corresponds to the predicates `debug/0`, `nodebug/0`, `trace/0`, `notrace/0`, `zip/0`, `nozip/0` (see Section 7.14 [Debug Pred], page 120).

    `trace`
    : Turns on trace mode.

    `debug`
    : Turns on the debugger.

`off` Turns off trace mode and the debugger (the default).

(This flag is not available in Runtime Systems.)

`character_escapes`
`on` or `off`. If this flag is `on`, a backslash occurring inside integers in `0''` notation or inside quoted atoms or strings has special meaning, and indicates the start of an escape sequence (see [Escape Sequences], page 401). This flag is relevant when reading as well as when writing terms, and is initially `on`.

`fileerrors`
`on` or `off`. Turns raising of exception on file errors on or off. Equivalent to `fileerrors/0` and `nofileerrors/0`, respectively (see Section 7.1.5 [Stream Pred], page 89). Initially `on`.

`gc` `on` or `off`. Turns garbage collection of the global stack on or off. Initially `on`.

`gc_margin`
Margin: Number of kilobytes. If less than *Margin* kilobytes are reclaimed in a garbage collection of the global stack then the size of the global stack should be increased. Also, no garbage collection is attempted unless the global stack is at least *Margin* kilobytes. Initially 500.

`gc_trace` Governs global stack garbage collection trace messages.

`verbose` Turn on verbose tracing of garbage collection.

`terse` Turn on terse tracing of garbage collection.

`off` Turn off tracing of garbage collection (the default).

`redefine_warnings`
`on` or `off`. Enable or disable warning messages when a predicate is being

- redefined from a different file than its previous definition.
- imported to the `user` module and it was previously locally defined.
- redefined locally and it was previously imported.
- imported to the `user` module from another module than it was previously imported from.

Initially `on`. (This warning is always disabled in Runtime Systems.)

`single_var_warnings`
`on` or `off`. Enable or disable warning messages when a clause containing variables not beginning with _ occurring once only is compiled or consulted. Initially `on`.

`unknown` Corresponds to the predicate `unknown/2` (see Section 7.14 [Debug Pred], page 120).

`trace` Causes calls to undefined predicates to be reported and the debugger to be entered at the earliest opportunity. (This setting is not possible in Runtime Systems.)

`fail` Causes calls to such predicates to fail.

`error` Causes calls to such predicates to raise an exception (the default). See Section 7.5 [Exception], page 101.

`syntax_errors`
Controls what action is taken upon syntax errors in `read/(1-2)`.

`dec10` The syntax error is reported and the read is repeated.

`error` An exception is raised. See Section 7.5 [Exception], page 101. (the default).

`fail` The syntax error is reported and the read fails.

`quiet` The read quietly fails.

`system_type`
A read-only flag. The value is `development` in Development Systems and `runtime` in Runtime Systems.

`typein_module`
Permitted values are atoms. Controls the current type-in module (see Section 4.2 [Module Spec], page 48). Corresponds to the predicate `module/1`.

`user_input`
Permitted values are any stream opened for reading. Controls which stream is referenced by `user_input` and `SP_stdin`. It is initially set to a stream connected to UNIX `stdin`.

`user_output`
Permitted values are any stream opened for writing. Controls which stream is referenced by `user_output` and `SP_stdout`. It is initially set to a stream connected to UNIX `stdout`.

`user_error`
Permitted values are any stream opened for writing. Controls which stream is referenced by `user_error` and `SP_stderr`. It is initially set to a stream connected to UNIX `stderr`.

`version` A read-only flag. The value is an atom containing the banner text displayed on startup and reinitialization, such as `'SICStus 3 #0: Wed Mar 15 12:29:29 MET 1995'`.

`toplevel_print_options`
The value is a list of options for `write_term/3` (see Section 7.1.3 [Term I/O], page 80), to be used when the top level displays variable bindings, answer constraints, and uncaught exceptions. Not available in Runtime Systems. The initial value is `[quoted(true),numbervars(true),portrayed(true),max_depth(10)]`.

`debugger_print_options`
: The value is a list of options for `write_term/3` (see Section 7.1.3 [Term I/O], page 80), to be used in the debugger's messages. Not available in Runtime Systems. The initial value is `[quoted(true),numbervars(true),portrayed(true),max_depth(10)]`.

`source_info`
: `on` or `off`. If `on` while code is being loaded, information about line numbers and filenames are stored with the loaded code. If `on` while debugging, this information is used to display the source code location while prompting for a debugger command. Initially `off`. Currently, this facility is only useful with the GNU Emacs interface.

`prolog_flag(`*?FlagName*`,`*?Value*`)`
: *Value* is the current value of the Prolog flag *FlagName*. Can be used to enumerate all Prolog flags and their values by backtracking.

`prolog_load_context(`*?Key*`,`*?Value*`)`
: This predicate gives access to context variables during compilation and loading of Prolog files. It unifies *Value* with the value of the variable identified by *Key*. Possible keys are:

  `file` The absolute path name of the file being compiled. During loading of a ql-file, the corresponding source file name is returned.

  `directory`
  : The absolute path name of the directory of the file being compiled/loaded.

  `module` The source module (see Section 4.5 [Meta Exp], page 49). This is useful for example if you are defining clauses for `user:term_expansion/(2,4)` and need to access the source module at compile time.

  `stream` The stream being compiled or loaded from.

  `term_position`
  : A term representing the position of the last clause read (see Section 7.1.5 [Stream Pred], page 89).

`statistics`
: Displays on the standard error stream statistics relating to memory usage, run time, garbage collection of the global stack and stack shifts.

`statistics(`*?Key*`,`*?Value*`)`
: This allows a program to gather various execution statistics. For each of the possible keys *Key*, *Value* is unified with a list of values, as follows:

  `global_stack`
  : [*size used*`,`*free*]
  This refers to the global stack, where compound terms are stored. The values are gathered before the list holding the answers is allocated.

`local_stack`
: [*size used*,*free*]
This refers to the local stack, where recursive predicate environments are stored.

`trail` [*size used*,*free*]
: This refers to the trail stack, where conditional variable bindings are recorded.

`choice` [*size used*,*free*]
: This refers to the choicepoint stack, where partial states are stored for backtracking purposes.

`core`
`memory` [*size used*,`0`]
: These refer to the amount of memory actually allocated by the process.

`heap`
`program` [*size used*,`0`]
: These refer to the amount of memory allocated for compiled and interpreted clauses, symbol tables, and the like.

`runtime` [*since start of Prolog*,*since previous statistics*] These refer to CPU time used while executing, excluding time spent garbage collecting, stack shifting, or in system calls. In Muse, these numbers refer to the worker that happens to be executing the call to `statistics/2`, and so normally are not meaningful.

`walltime` [*since start of Prolog*,*since previous statistics*] These refer to absolute time elapsed.

`garbage_collection`
: [*no. of GCs*,*bytes freed*,*time spent*]

`stack_shifts`
: [*no. of local shifts*,*no. of trail shifts*,*time spent*]

`atoms` [*no. of atoms*,*bytes used*,*bytes free*]

`atom_garbage_collection`
: [*no. of AGCs*,*bytes freed*,*time spent*]

Times are in milliseconds, sizes of areas in bytes.

7.7 Meta-Logic

The predicates in this section are meta-logical and perform operations that require reasoning about the current instantiation of terms or decomposing terms into their constituents. Such operations cannot be expressed using predicate definitions with a finite number of clauses.

`var(?`*X*`)` *[ISO]*
: Tests whether *X* is currently uninstantiated (`var` is short for variable). An uninstantiated variable is one which has not been bound to anything, except possibly another uninstantiated variable. Note that a structure with some components which are uninstantiated is not itself considered to be uninstantiated. Thus the directive

```
| ?- var(foo(X, Y)).
```

always fails, despite the fact that *X* and *Y* are uninstantiated.

`nonvar(?`*X*`)` *[ISO]*
: Tests whether *X* is currently instantiated. This is the opposite of `var/1`.

`ground(?`*X*`)`
: Tests whether *X* is completely instantiated, i.e. free of unbound variables. In this context, mutable terms are treated as nonground, so as to make `ground/1` a monotone predicate.

`atom(?`*X*`)` *[ISO]*
: Checks that *X* is currently instantiated to an atom (i.e. a non-variable term of arity 0, other than a number).

`float(?`*X*`)` *[ISO]*
: Checks that *X* is currently instantiated to a float.

`integer(?`*X*`)` *[ISO]*
: Checks that *X* is currently instantiated to an integer.

`number(?`*X*`)` *[ISO]*
: Checks that *X* is currently instantiated to a number.

`atomic(?`*X*`)` *[ISO]*
: Checks that *X* is currently instantiated to an atom or number.

`simple(?`*X*`)`
: Checks that *X* is currently uninstantiated or instantiated to an atom or number.

`compound(?`*X*`)` *[ISO]*
: Checks that *X* is currently instantiated to a term of arity `>` 0 i.e. a list or a structure.

`callable(?`*X*`)`
: Checks that *X* is currently instantiated to a term valid as a goal i.e. a compound term or an atom.

`is_mutable(?`*X*`)`
: Checks that *X* is currently instantiated to a mutable term (see Section 7.8 [Modify Term], page 113).

`functor(`+*Term*`,`*?Name*`,`*?Arity*`)` *[ISO]*
`functor(`*?Term*`,`+*Name*`,`+*Arity*`)` *[ISO]*

The principal functor of term *Term* has name *Name* and arity *Arity*, where *Name* is either an atom or, provided *Arity* is 0, a number. Initially, either *Term* must be instantiated, or *Name* and *Arity* must be instantiated to, respectively, either an atom and an integer in [0,255] or an atomic term and 0. In the case where *Term* is initially uninstantiated, the result of the call is to instantiate *Term* to the most general term having the principal functor indicated.

`arg(`+*ArgNo*`,`+*Term*`,`*?Arg*`)` *[ISO]*

Arg is the argument *ArgNo* of the compound term *Term*. The arguments are numbered from 1 upwards, *ArgNo* must be instantiated to a positive integer and *Term* to a compound term.

+*Term* `=..` *?List [ISO]*
?Term `=..` +*List [ISO]*

List is a list whose head is the atom corresponding to the principal functor of *Term*, and whose tail is a list of the arguments of *Term*. e.g.

```
| ?- product(0, n, n-1) =.. L.

L = [product,0,n,n-1]

| ?- n-1 =.. L.

L = [-,n,1]

| ?- product =.. L.

L = [product]
```

If *Term* is uninstantiated, then *List* must be instantiated either to a list of determinate length whose head is an atom, or to a list of length 1 whose head is a number. Note that this predicate is not strictly necessary, since its functionality can be provided by `arg/3` and `functor/3`, and using the latter two is usually more efficient.

`name(`+*Const*`,`*?CharList*`)`
`name(`*?Const*`,`+*CharList*`)`

If *Const* is an atom or number, *CharList* is a list of the character codes of the characters comprising the name of *Const*. e.g.

```
| ?- name(product, L).

L = [112,114,111,100,117,99,116]

| ?- name(product, "product").

| ?- name(1976, L).

L = [49,57,55,54]
```

```
| ?- name('1976', L).

L = [49,57,55,54]

| ?- name((:-), L).

L = [58,45]
```

If *Const* is uninstantiated, *CharList* must be instantiated to a list of character codes. If *CharList* can be interpreted as a number, *Const* is unified with that number, otherwise with the atom whose name is *CharList*. E.g.

```
| ?- name(X, [58,45]).

X = :-

| ?- name(X, ":-").

X = :-

| ?- name(X, [49,50,51]).

X = 123
```

Note that there atoms are for which `name(`*Const*`,`*CharList*`)` is true, but which will not be constructed if `name/2` is called with *Const* uninstantiated. One such atom is the atom `'1976'`. It is recommended that new programs use `atom_chars/2` or `number_chars/2`, as these predicates do not have this inconsistency.

`atom_chars(+`*Const*`,`*?CharList*`)` *[ISO]*
`atom_chars(`*?Const*`,+`*CharList*`)` *[ISO]*

The same as `name(`*Const*`,`*CharList*`)`, but *Const* is constrained to be an atom.

This predicate is called `atom_codes/2` in the ISO Prolog standard.

`number_chars(+`*Const*`,`*?CharList*`)` *[ISO]*
`number_chars(`*?Const*`,+`*CharList*`)` *[ISO]*

The same as `name(`*Const*`,`*CharList*`)`, but *Const* is constrained to be a number.

This predicate is called `number_codes/2` in the ISO Prolog standard.

`copy_term(`*?Term*`,`*?CopyOfTerm*`)` *[ISO]*

CopyOfTerm is a renaming of *Term*, such that brand new variables have been substituted for all variables in *Term*. If any of the variables of *Term* have goals blocked on them, the copied variables will have copies of the goals blocked on them as well. Similarly, independent copies are substituted for any mutable terms in *term*. It behaves as if defined by:

```
copy_term(X, Y) :-
        assert('copy of'(X)),
        retract('copy of'(Y)).
```

The implementation of `copy_term/2` conserves space by not copying ground subterms.

7.8 Modification of Terms

One of the tenets of logic programming is that terms are immutable objects of the Herbrand universe, and the only sense in which they can be modified is by means of instantiating non-ground parts. There are, however, algorithms where destructive assignment is essential for performance. Although alien to the ideals of logic programming, this feature can be defended on practical grounds.

SICStus Prolog provides an abstract datatype and three operations for efficient *backtrackable* destructive assignment. In other words, any destructive assignments are transparently undone on backtracking. Modifications that are intended to survive backtracking must be done by asserting or retracting dynamic program clauses instead. Unlike previous releases of SICStus Prolog, destructive assignment of arbitrary terms is not allowed.

A *mutable term* is represented as a compound terms with a reserved functor: `'$mutable'(`*Value*`,`*Timestamp*`)` where *Value* is the current value and *Timestamp* is reserved for bookkeeping purposes.

Any copy of a mutable term created by `copy_term/2`, `assert`, `retract`, an internal database predicate, or an all solutions predicate, is an independent copy of the original mutable term. Any destructive assignment done to one of the copies will not affect the other copy.

The following operations are provided:

`create_mutable(+`*Datum*`,-`*Mutable*`)`
: *Mutable* is a new mutable term with initial value *Datum*. *Datum* must not be an unbound variable.

`get_mutable(?`*Datum*`,+`*Mutable*`)`
: *Datum* is the current value of the mutable term *Mutable*.

`update_mutable(+`*Datum*`,+`*Mutable*`)`
: Updates the current value of the mutable term *Mutable* to become *Datum*. *Datum* must not be an unbound variable.

`is_mutable(?`*Mutable*`)`
: Checks that *Mutable* is currently instantiated to a mutable term (see Section 7.8 [Modify Term], page 113).

7.9 Modification of the Program

The predicates defined in this section allow modification of dynamic predicates. Dynamic clauses can be added (*asserted*) or removed from the program (*retracted*).

For these predicates, the argument *Head* must be instantiated to an atom or a compound term, with an optional module prefix. The argument *Clause* must be instantiated either to a term *Head* `:-` *Body* or, if the body part is empty, to *Head*, with an optional module prefix. An empty body part is represented as `true`.

Note that a term *Head* `:-` *Body* must be enclosed in parentheses when it occurs as an argument of a compound term, as '`:-`' is a standard infix operator with precedence greater than 1000 (see Section 2.6 [Operators], page 37), e.g.:

```
| ?- assert((Head :- Body)).
```

Like recorded terms (see Section 7.10 [Database], page 115), the clauses of dynamic predicates have a unique implementation-defined identifier. Some of the predicates below have an additional argument which is this identifier. This identifier makes it possible to access clauses directly instead of requiring a normal database (hash-table) lookup.

`assert(`*:Clause*`)`
`assert(`*:Clause*`,-`*Ref*`)`
: The current instance of *Clause* is interpreted as a clause and is added to the current interpreted program. The predicate concerned must currently be dynamic or undefined and the position of the new clause within it is implementation-defined. *Ref* is a unique identifier of the asserted clause. Any uninstantiated variables in the *Clause* will be replaced by new private variables, along with copies of any subgoals blocked on these variables (see Section 2.3 [Procedural], page 34).

`asserta(`*:Clause*`)` *[ISO]*
`asserta(`*:Clause*`,-`*Ref*`)`
: Like `assert/2`, except that the new clause becomes the *first* clause for the predicate concerned.

`assertz(`*:Clause*`)` *[ISO]*
`assertz(`*:Clause*`,-`*Ref*`)`
: Like `assert/2`, except that the new clause becomes the *last* clause for the predicate concerned.

`clause(`*:Head*`,`*?Body*`)` *[ISO]*
`clause(`*:Head*`,`*?Body*`,`*?Ref*`)`
`clause(`*?Head*`,`*?Body*`,+`*Ref*`)`
: The clause `(`*Head* `:-` *Body*`)` exists in the current interpreted program, and is uniquely identified by *Ref*. The predicate concerned must currently be dynamic. At the time of call, either *Ref* must be instantiated to a valid identifier, or *Head* must be instantiated to an atom or a compound term. Thus `clause/3` can have two different modes of use.

`retract(`*:Clause*`)` *[ISO]*
: The first clause in the current interpreted program that matches *Clause* is erased. The predicate concerned must currently be dynamic. `retract/1` may be used in a non-determinate fashion, i.e. it will successively retract clauses matching the argument through backtracking. If reactivated by backtracking, invocations of the predicate whose clauses are being retracted will proceed unaffected by the retracts. This is also true for invocations of `clause/(2-3)` for the same predicate. The space occupied by a retracted clause will be recovered when instances of the clause are no longer in use.

`retractall(`*:Head*`)`
: Erases all clauses whose head matches *Head*, where *Head* must be instantiated to an atom or a compound term. The predicate concerned must currently be dynamic. The predicate definition is retained.

Note: all predicates mentioned above first look for a predicate that is visible in the module in which the call textually appears. If no predicate is found, a new dynamic predicate (with no clauses) is created automatically. During OR-parallel execution, however, predicates must not be defined on the fly like this. It is recommended to declare as dynamic predicates for which clauses will be asserted.

`abolish(`*:Spec*`)` *[ISO]*
`abolish(`*:Name*`,+`*Arity*`)`
: Erases all clauses of the predicate specified by *Spec* or *Name/Arity*. *Spec* has the same form as for `spy/1` (see Section 6.3 [Spy-Point], page 62) and *Name* may be prefixed by a module name (see Section 4.2 [Module Spec], page 48).

  The predicate definition and all associated information such as spy-points is also erased. The predicates concerned must all be user defined.

`erase(+`*Ref*`)`
: The dynamic clause or recorded term (see Section 7.10 [Database], page 115) whose implementation-defined identifier is *Ref* is effectively erased from the internal database or interpreted program.

`instance(+`*Ref*`,?`*Term*`)`
: A (most general) instance of the dynamic clause or recorded term whose implementation-defined identifier is *Ref* is unified with *Term*. *Ref* must be instantiated to a legal identifier.

7.10 Internal Database

The predicates described in this section were introduced in early implementations of Prolog to provide efficient means of performing operations on large quantities of data. The introduction of indexed dynamic predicates have rendered these predicates obsolete, and the sole purpose of providing them is to support existing code. There is no reason whatsoever to use them in new code.

These predicates store arbitrary terms in the database without interfering with the clauses which make up the program. The terms which are stored in this way can subsequently be retrieved via the key on which they were stored. Many terms may be stored on the same key, and they can be individually accessed by pattern matching. Alternatively, access can be achieved via a special identifier which uniquely identifies each recorded term and which is returned when the term is stored.

`recorded(`*?Key*`,`*?Term*`,`*?Ref*`)`
: The internal database is searched for terms recorded under the key *Key*. These terms are successively unified with *Term* in the order they occur in the database. At the same time, *Ref* is unified with the implementation-defined identifier uniquely identifying the recorded item. If the key is instantiated to a compound term, only its principal functor is significant. If the key is uninstantiated, all terms in the database are successively unified with *Term* in the order they occur.

`recorda(+`*Key*`,`*?Term*`,-`*Ref*`)`
: The term *Term* is recorded in the internal database as the first item for the key *Key*, where *Ref* is its implementation-defined identifier. The key must be given, and only its principal functor is significant. Any uninstantiated variables in the *Term* will be replaced by new private variables, along with copies of any subgoals blocked on these variables (see Section 2.3 [Procedural], page 34).

`recordz(+`*Key*`,`*?Term*`,-`*Ref*`)`
: Like `recorda/3`, except that the new term becomes the *last* item for the key *Key*.

`current_key(`*?KeyName*`,`*?KeyTerm*`)`
: *KeyTerm* is the most general form of the key for a currently recorded term, and *KeyName* is the name of that key. This predicate can be used to enumerate in undefined order all keys for currently recorded terms through backtracking.

7.11 Blackboard Primitives

The predicates described in this section store arbitrary terms in a per-module repository known as the "blackboard". The main purpose of the blackboard is to provide a means for communication between branches executing in parallel, but the blackboard works equally well during sequential execution. The blackboard implements a mapping from keys to values. Keys are restricted to being atoms or integers in the range `[-33554432, 33554431]`, whereas values are arbitrary terms. In contrast to the predicates described in the previous sections, a given key can map to at most a single term.

Each Prolog module maintains its own blackboard, so as to avoid name clashes if different modules happen to use the same keys. The "key" arguments of these predicates are subject to module name expansion, so the module name does not have to be explicitly given unless multiple Prolog modules are supposed to share a single blackboard.

The predicates below implement atomic blackboard actions. In Muse, these predicates are *cavalier* (see Section 3.5 [Programming Considerations], page 45).

`bb_put(:`*Key*`, +`*Term*`)`
: A copy of *Term* is stored under *Key*. Any previous term stored under the same *Key* is simply deleted.

`bb_get(:`*Key*`, ?`*Term*`)`
: If a term is currently stored under *Key*, a copy of it is unified with *Term*. Otherwise, `bb_get/2` silently fails.

`bb_delete(`*:Key*`, `*?Term*`)`
: If a term is currently stored under *Key*, the term is deleted, and a copy of it is unified with *Term*. Otherwise, `bb_delete/2` silently fails.

`bb_update(`*:Key*`, `*?OldTerm*`, `*?NewTerm*`)`
: If a term is currently stored under *Key* and unifies with *OldTerm*, the term is replaced by a copy of *NewTerm*. Otherwise, `bb_update/3` silently fails. This predicate provides an atomic swap operation.

The following example illustrates how these primitives may be used to implement a "maxof" predicate that finds the maximum value computed by some non-deterministic goal, which may execute in parallel. We use a single key `max`. Note the technique of using `bb_update/3` in a repeat-fail loop, since other execution branches may be competing for updating the value, and we only want to store a new value if it is greater than the old value.

We assume that *Goal* does not produce any "false" solutions that would be eliminated by cuts in a sequential execution. Thus, *Goal* may need to include redundant checks to ensure that its solutions are valid, as discussed above.

```
maxof(Value, Goal, _) :-
        bb_put(max, -1),                % initialize max-so-far
        call(Goal),
        update_max(Value),
        fail.
maxof(_, _, Max) :-
        bb_delete(max, Max),
        Max > 1.

update_max(New):-
        repeat,
          bb_get(max, Old),
          compare(C, Old, New),
          update_max(C, Old, New), !.

update_max(<, Old, New) :- bb_update(max, Old, New).
update_max(=, _, _).
update_max(>, _, _).
```

7.12 All Solutions

When there are many solutions to a problem, and when all those solutions are required to be collected together, this can be achieved by repeatedly backtracking and gradually building up a list of the solutions. The following built-in predicates are provided to automate this process.

Note that the *Goal* argument to the predicates listed below is called as if by `call/1` at runtime. Thus if *Goal* is complex and if performance is an issue, define an auxiliary predicate which can then be compiled, and let *Goal* call it.

`setof(`*?Template*`,`*:Goal*`,`*?Set*`)` *[ISO]*

Read this as "*Set* is the set of all instances of *Template* such that *Goal* is satisfied, where that set is non-empty". The term *Goal* specifies a goal or goals as in `call(`*Goal*`)` (see Section 7.4 [Control], page 99). *Set* is a set of terms represented as a list of those terms, without duplicates, in the standard order for terms (see Section 7.3 [Term Compare], page 97). If there are no instances of *Template* such that *Goal* is satisfied then the predicate fails.

The variables appearing in the term *Template* should not appear anywhere else in the clause except within the term *Goal*. Obviously, the set to be enumerated should be finite, and should be enumerable by Prolog in finite time. It is possible for the provable instances to contain variables, but in this case the list *Set* will only provide an imperfect representation of what is in reality an infinite set.

If there are uninstantiated variables in *Goal* which do not also appear in *Template*, then a call to this built-in predicate may backtrack, generating alternative values for *Set* corresponding to different instantiations of the free variables of *Goal*. (It is to cater for such usage that the set *Set* is constrained to be non-empty.) Two instantiations are different iff no renaming of variables can make them literally identical. For example, given the clauses:

```
likes(bill, cider).
likes(dick, beer).
likes(harry, beer).
likes(jan, cider).
likes(tom, beer).
likes(tom, cider).
```

the query

```
| ?- setof(X, likes(X,Y), S).
```

might produce two alternative solutions via backtracking:

```
S = [dick,harry,tom],
Y = beer ? ;

S = [bill,jan,tom],
Y = cider ? ;
```

The query:

```
| ?- setof((Y,S), setof(X, likes(X,Y), S), SS).
```

would then produce:

```
SS = [(beer,[dick,harry,tom]),(cider,[bill,jan,tom])]
```

Variables occurring in *Goal* will not be treated as free if they are explicitly bound within *Goal* by an existential quantifier. An existential quantification is written:

```
Y^Q
```

meaning "there exists a Y such that Q is true", where Y is some Prolog variable.

For example:

```
| ?- setof(X, Y^(likes(X,Y)), S).
```

would produce the single result:

```
S = [bill,dick,harry,jan,tom]
```

in contrast to the earlier example.

`bagof(`*?Template*`,`*:Goal*`,`*?Bag*`)` *[ISO]*
: This is exactly the same as `setof/3` except that the list (or alternative lists) returned will not be ordered, and may contain duplicates. The effect of this relaxation is to save a call to `sort/2`, which is invoked by `setof/3` to return an ordered list.

?X^:P
: The all solution predicates recognize this as meaning "there exists an X such that P is true", and treats it as equivalent to P (see Section 7.4 [Control], page 99). The use of this explicit existential quantifier outside the `setof/3` and `bagof/3` constructs is superfluous and discouraged.

`findall(`*?Template*`,`*:Goal*`,`*?Bag*`)` *[ISO]*
: *Bag* is a list of instances of *Template* in all proofs of *Goal* found by Prolog. The order of the list corresponds to the order in which the proofs are found. The list may be empty and all variables are taken as being existentially quantified. This means that each invocation of `findall/3` succeeds *exactly once*, and that no variables in *Goal* get bound. Avoiding the management of universally quantified variables can save considerable time and space.

`findall(`*?Template*`,`*:Goal*`,`*?Bag*`,`*?Remainder*`)`
: Same as `findall/3`, except *Bag* is the list of solution instances appended with *Remainder*, which is typically unbound.

7.13 Coroutining

The coroutining facility can be accessed by a number of built-in predicates. This makes it possible to use coroutines in a dynamic way, without having to rely on block declarations:

`when(+`*Condition*`,`*:Goal*`)`
: Blocks *Goal* until the *Condition* is true, where *Condition* is a Prolog goal with the restricted syntax:

  `nonvar(`X`)`

  `ground(`X`)`

  `?=(`X`,`Y`)`

  Condition`,`*Condition*

  Condition`;`*Condition*

  For example:

```
| ?- when(((nonvar(X);?=(X,Y)),ground(T)), process(X,Y,T)).
```

`freeze(`*?X*`,`*:Goal*`)`
: Blocks *Goal* until `nonvar(`X`)` (see Section 7.7 [Meta Logic], page 110) holds. This is defined as if by:

```
freeze(X, Goal) :- when(nonvar(X), Goal).
```

  or

```
:- block freeze(-, ?).
freeze(_, Goal) :- Goal.
```

`frozen(`-*Var*`,` *?Goal*`)`
: If some goal is blocked on the variable *Var*, or *Var* has attributes that can be interpreted as a goal (see Chapter 14 [Attributes], page 193), then that goal is unified with *Goal*. If no goals are blocked, *Goal* is unified with the atom `true`. If more than one goal is blocked, a conjunction is unified with *Goal*.

`dif(`*?X*`,`*?Y*`)`
: Constrains X and Y to represent different terms i.e. to be non-unifiable. Calls to `dif/2` either succeed, fail, or are blocked depending on whether X and Y are sufficiently instantiated. It is defined as if by:

```
dif(X, Y) :- when(?=(X,Y), X\==Y).
```

`call_residue(`:*Goal*`,` *?Residue*`)`
: The *Goal* is executed as if by `call/1`. If after the execution there are still some subgoals of *Goal* that are blocked on some variables, then *Residue* is unified with a list of *VariableSet*-*Goal* pairs, and those subgoals are no longer blocked on any variables. Otherwise, *Residue* is unified with the empty list `[]`.

  VariableSet is a set of variables such that when any of the variables is bound, *Goal* gets unblocked. Usually, a goal is blocked on a single variable, in which case *VariableSet* is a singleton.

  Goal is an ordinary goal, sometimes module prefixed. For example:

```
| ?- call_residue((dif(X,f(Y)), X=f(Z)), Res).

X = f(Z),
Res = [[Y,Z]-(prolog:dif(f(Z),f(Y)))]
```

7.14 Debugging

Debugging predicates are not available in Runtime Systems.

`unknown(`*?OldState*`,` *?NewState*`)`
: *OldState* is the current state of the "Action on unknown predicates" flag, and sets the flag to *NewState*. This flag determines whether or not the system is to catch calls to undefined predicates (see Section 1.6 [Undefined Predicates], page 15), when `user:unknown_predicate_handler/3` cannot handle the goal. The possible states of the flag are:

  `trace`
  : Causes calls to undefined predicates to be reported and the debugger to be entered at the earliest opportunity.

  `fail`
  : Causes calls to such predicates to fail.

  `error`
  : Causes calls to such predicates to raise an exception (the default). See Section 7.5 [Exception], page 101.

`debug`
: The debugger is switched on in *leap* mode. See Section 6.2 [Basic Debug], page 61.

`trace` The debugger is switched on in *creep* mode. See Section 6.2 [Basic Debug], page 61.

`zip` The debugger is switched on in *zip* mode. See Section 6.2 [Basic Debug], page 61.

`nodebug`
`notrace`
`nozip` The debugger is switched off. See Section 6.2 [Basic Debug], page 61.

`leash(+`*Mode*`)`
Leashing Mode is set to *Mode*. See Section 6.2 [Basic Debug], page 61.

`spy` *:Spec* Spy-points are placed on all the predicates given by *Spec*. See Section 6.3 [Spy-Point], page 62.

`nospy` *:Spec*
Spy-points are removed from all the predicates given by *Spec*. See Section 6.3 [Spy-Point], page 62.

`nospyall` Removes all the spy-points that have been set.

`debugging`
Displays information about the debugger. See Section 6.2 [Basic Debug], page 61.

7.15 Execution Profiling

Execution profiling is a common aid for improving software performance. The SICStus Prolog compiler has the capability of instrumenting compiled code with *counters* which are initially zero and incremented whenever the flow of control passes a given point in the compiled code. This way the number of calls, backtracks, choicepoints created, etc., can be counted for the instrumented predicates, and an estimate of the time spent in individual clauses and disjuncts can be calculated.

Gauge is a graphical user interface for inspecting execution profiles. It is available as a library module (see Chapter 35 [Gauge Intro], page 367).

The original version of the profiling package was written by M.M. Gorlick and C.F. Kesselman at the Aerospace Corporation [Gorlick & Kesselman 87].

Profiling is not available in Muse.

Only compiled code can be instrumented. To get an execution profile of a program, the compiler must first be told to produce instrumented code. This is done by issuing the directive:

```
| ?- prolog_flag(compiling,_,profiledcode).
```

after which the program to be analyzed can be compiled as usual. Any new compiled code will be instrumented while the compilation mode flag has the value `profiledcode`.

The profiling data is generated by simply running the program. The predicate `profile_data/4` (see below) makes available a selection of the data as a Prolog term. The predicate `profile_reset/1` zeroes the profiling counters for a selection of the currently instrumented predicates.

`profile_data(`*:Spec*`,`*?Selection*`,`*?Resolution*`,`*-Data*`)`

Data is profiling data collected from the predicates covered by *Spec*, which has the same form as for `spy/1` (see Section 6.3 [Spy-Point], page 62).

The *Selection* argument determines the kind of profiling data to be collected. If uninstantiated, the predicate will backtrack over its possible values, which are:

`calls` All instances of entering a clause by a procedure call are counted. This is equivalent to counting all procedure calls *that have not been determined to fail by indexing on the first argument.*

`backtracks`
All instances of entering a clause by backtracking are counted.

`choice_points`
All instances of creating a choicepoint are counted. This occurs, roughly, when the implementation determines that there are more than one possibly matching clauses for a procedure call, and when a disjunction is entered.

`shallow_fails`
Failures in the "if" part of if-then-else statements, and in the "guard" part of guarded clauses, are counted as *shallow failures*. See Section 10.4 [If-Then-Else], page 178.

`deep_fails`
Any failures that do not classify as shallow as above are counted as *deep failures*. The reason for distinguishing shallow and deep failures is that the former are considerably cheaper to execute than the latter.

`execution_time`
The execution time for the selected predicates, clauses, or disjuncts is estimated in artificial units.

The *Resolution* argument determines the level of resolution of the profiling data to be collected. If uninstantiated, the predicate will backtrack over its possible values, which are:

`predicate`
Data is a list of *Module:PredName-Count*, where *Count* is a sum of the corresponding counts per clause.

`clause` *Data* is a list of *Module:ClauseName-Count*, where *Count* includes counts for any disjunctions occurring inside that clause. Note, however, that the selections `calls` and `backtracks` do *not* include counts for disjunctions.

`all` *Data* is a list of *Module:InternalName-Count*. This is the finest resolution level, counting individual clauses and disjuncts.

Above, *PredName* is a predicate spec, *ClauseName* is a compound term *PredName/ClauseNumber*, and *InternalName* is either
ClauseName—corresponding to a clause, or
`(`*ClauseName-DisjNo*`)/`*Arity*`/`*AltNo*—corresponding to a disjunct.

`profile_reset(`*:Spec*`)`

Zeroes all counters for predicates covered by *Spec*, which has the same form as for `spy/1` (see Section 6.3 [Spy-Point], page 62).

7.16 Muse

The following predicates are only defined in the Muse Development System.

`muse_sync`
: A built-in predicate that can be used to synchronize for being leftmost in the parallel search tree. Its main use is in combination with foreign language interface calls.

`muse_flags`
: This built-in predicate lists all current Muse flag settings.

`muse_flag(`*?Flag*`,-`*Old*`)`
`muse_flag(`*?Flag*`,-`*Old*`,+`*New*`)`
: These examine and/or adjust parts of the Muse scheduler state. The current value is returned in *Old*, and a new value may be specified in *New*.

  The state cannot be changed when Muse is executing parallel work. It is therefore recommended to perform such changes as the first (or best single) goal in a top level query. If used elsewhere, the execution may be aborted if the system cannot perform the change.

  Possible flag values are:

  `num_workers`
  : Returns in *Old* the current number of workers, which is adjusted to *New*.

  `max_workers`
  : The maximum number of Muse workers is returned. This flag is read-only.

  `worker_id`
  : The identity of the currently executing Muse workers is returned. This flag is read-only.

  `version` The Muse version is returned. The version can be `plain` or `trace`. This flag is read-only.

  The following flags also exist, but understanding them requires deep knowledge about the Muse scheduler and execution model, so they are probably not useful for the average user.

  `vol_susp` Used to turn voluntary suspension on (1) or off (0).

  `vol_susp_time`
  : Voluntary suspension interval, in milliseconds,

  `vol_susp_num`
  : Number of workers to voluntarily suspend.

  `vol_susp_avoid`
  : Controls the eagerness of voluntary suspension after resumption.

  `eager_cut`
  : Used to turn eager pruning on (1) or off (0).

  `max_sch_loops`
  : Maximum number of scheduler loops.

`max_suspended`
: Maximum number of megabytes for suspended branches.

`muse_trace(`*:Goal*`)`
`muse_trace(`*:Goal*`,+`*FilePrefix*`)`
: The trace version generates a trace file for the goal *Goal*. The file may be prefixed with *FilePrefix*. The file is created in the current working directory if it can be written there, or otherwise in the directory that is the value of the `TMPDIR` environment variable.

  Cannot be executed in parallel when the execution changes the scheduler state.

7.17 Miscellaneous

?X `=` *?Y* *[ISO]*
: Defined as if by the clause `Z=Z.`; i.e. X and Y are unified.

`length(`*?List*`,`*?Length*`)`
: If *List* is instantiated to a list of determinate length, then *Length* will be unified with this length.

  If *List* is of indeterminate length and *Length* is instantiated to an integer, then *List* will be unified with a list of length *Length*. The list elements are unique variables.

  If *Length* is unbound then *Length* will be unified with all possible lengths of *List*.

`numbervars(`*?Term*`,+`*N*`,`*?M*`)`
: Unifies each of the variables in term *Term* with a special term, so that `write(`*Term*`)` (or `writeq(`*Term*`)`) (see Section 7.1.3 [Term I/O], page 80) prints those variables as (`A +` (i mod 26))(i/26) where i ranges from N to M-1. N must be instantiated to an integer. If it is 0 you get the variable names A, B, ..., Z, A1, B1, etc. This predicate is used by `listing/(0-1)` (see Section 7.6 [State Info], page 104).

`halt` *[ISO]*
: Causes Prolog to exit back to the shell. (In recursive calls to Prolog from C, this predicate will return back to C instead.)

`halt(+`*Code*`)` *[ISO]*
: Causes the Prolog process to immediately exit back to the shell with the return code *Code*, even if it occurs in a recursive call from C.

`op(+`*Precedence*`,+`*Type*`,+`*Name*`)` *[ISO]*
: Declares the atom *Name* to be an operator of the stated *Type* and *Precedence* (see Section 2.6 [Operators], page 37). *Name* may also be a list of atoms in which case all of them are declared to be operators. If *Precedence* is 0 then the operator properties of *Name* (if any) are cancelled.

`current_op(`*?Precedence*`,`*?Type*`,`*?Op*`)` *[ISO]*
: The atom *Op* is currently an operator of type *Type* and precedence *Precedence*. Neither *Op* nor the other arguments need be instantiated at the time of the call; i.e. this predicate can be used to generate as well as to test.

`break` Invokes a recursive top-level. See Section 1.9 [Nested], page 16. (This predicate is not available in Runtime Systems nor in Muse.)

`abort` Aborts the current execution. See Section 1.9 [Nested], page 16. (In recursive calls to Prolog from C, this predicate will return back to C instead.) (In Muse, will adjust the system to one worker.)

`save_program(+`*File*`)`
`save_program(+`*File*`, :`*Goal*`)`
The system saves the program state into file *File*. When the program state is restored, *Goal* is executed. *Goal* defaults to `true`. See Section 1.10 [Saving], page 17. (Requires Muse to be adjusted to one worker.)

`restore(+`*File*`)`
The system is returned to the program state previously saved to file *File* with start-up goal *Goal*. `restore/1` may succeed, fail or raise an exception depending on *Goal*. See Section 1.10 [Saving], page 17. (Requires Muse to be adjusted to one worker.)

`reinitialise`
This predicate can be used to force the reinitialization behavior to take place at any time. When SICStus Prolog is reinitialized it:

- calls any user defined C function pointed at by `SP_reinit_hook` (see Section 8.7 [Hooks], page 150).
- calls `version/0` to write banners. %
- % looks for a file '`~/.sicstusrc`' or '`~/.sicstus.ini`' and % consults it, if it exists. %
- % ensures a file *File* is loaded if `-l` *File* was supplied as % program arguments.
- calls `initialization/0` to run user-defined initializations.

(In recursive calls to Prolog from C, this predicate will return back to C instead.) (In Muse, will adjust the system to one worker.)

`garbage_collect`
Performs a garbage collection of the global stack immediately.

`garbage_collect_atoms`
Performs a garbage collection of the atoms immediately.

`gc` Enables garbage collection of the global stack (the default).

`nogc` Disables garbage collection of the global stack.

`prompt(?`*Old*`,?`*New*`)`
The sequence of characters (prompt) which indicates that the system is waiting for user input is represented as an atom, and unified with *Old*; the atom bound to New specifies the new prompt. In particular, the goal `prompt(X, X)` unifies the current prompt with *X*, without changing it. Note that this predicate only affects the prompt given when a user's program is trying to read from the standard input stream (e.g. by calling `read/1`). Note also that the prompt is reset to the default '`| :` ' on return to top-level.

`version` Displays the introductory messages for all the component parts of the current system.

Prolog will display its own introductory message when initially run and on reinitialization by calling `version/0`. If this message is required at some other time it can be obtained using this predicate which displays a list of introductory messages; initially this list comprises only one message (Prolog's), but you can add more messages using `version/1`. (This predicate is not available in Runtime Systems.)

`version(+`*Message*`)`
Appends *Message* to the end of the message list which is output by `version/0`. *Message* must be an atom. (This predicate is not available in Runtime Systems.)

The idea of this message list is that, as systems are constructed on top of other systems, each can add its own identification to the message list. Thus `version/0` should always indicate which modules make up a particular package. It is not possible to remove messages from the list.

`initialization`
Executes goals defined by `initialization/1`. Only the first solution is investigated. `initialization/0` is called at system (re)initialization. (This predicate is not available in Runtime Systems.)

`initialization(:`*Goal*`)`
Appends *Goal* to the goal list which is executed by `initialization/0`. It is not possible to remove goals from the list. (This predicate is not available in Runtime Systems.)

`initialization/(0-1)` provide a mechanism which is similar but more general than `version/(0-1)`. It can, for example, be used by systems constructed on top of SICStus Prolog to load their own initialization files.

`help` *Hookable*, displays basic information, or a user defined help message. It first calls `user:user_help/0`, and only if that call fails is a default help message printed on the current output stream. (This predicate is not available in Runtime Systems.)

`user_help`
`user:user_help`
A hook predicate. This may be defined by the user to print a help message on the current output stream.

8 Mixing C and Prolog

SICStus Prolog provides a bi-directional, procedural interface for program parts written in C and Prolog. The C side of the interface defines a number of functions and macros for various operations. On the Prolog side, you have to supply declarations specifying the names and argument/value types of C functions being called as Prolog predicates. These declarations are used by the predicate `load_foreign_resource/1`, which performs the actual binding of C functions to Prolog predicates.

In most cases, the argument/value type declaration suffice for making the necessary conversions of data automatically as they are passed between C and Prolog. However, it is possible to declare the type of an argument to be a Prolog term, in which case the receiving function will see it as a "handle" object, called an `SP_term_ref`, for which access functions are provided.

The C support routines are available in a Development System as well as in Runtime Systems. The support routines include:

- Static and dynamic linking of C code into the Prolog environment.
- Automatic conversion between Prolog terms and C data with `foreign/(2-3)` declarations.
- Functions for accessing and creating Prolog terms, and for creating and manipulating SP_term_refs.
- The Prolog system may call C predicates which may call Prolog back without limits on recursion.
- Possibility to create stand-alone applications, *Runtime Systems*.
- Possibility to create user defined Prolog streams.
- Functions to read and write on Prolog streams from C.
- Possibility to install interrupt handlers that can safely call Prolog.
- User hooks that can be used to perform user defined actions on a number of occasions e.g. before reading a character from the standard input stream, upon reinitialization, etc.

8.1 Notes

ANSI Conformance
: Throughout this chapter, `void *` in the function definitions may be changed to `char *` on non ANSI conforming C compilers.

The SP_PATH variable
: It is normally not necessary to set this environment variable, but its value will be used at runtime if no explicit boot path is given when initializing a Runtime or Development System. In this chapter, the environment variable `SP_PATH` is used as a shorthand for the SICStus Prolog installation directory, whose default UNIX location is '`/usr/local/lib/sicstus37`'). See Section 1.1.2 [Environment Variables], page 10.

Definitions and declarations
: Type definitions and function declarations for the interface are found in the header file '`<sicstus/sicstus.h>`'.

Error Codes

The value of many support functions is a return code which is one of `SP_SUCCESS` for success, `SP_FAILURE` for failure, `SP_ERROR` if an error condition occurred. In particular, uncaught exceptions resulting from calls from C to Prolog raise an error condition. In error situations, the macro `SP_errno` will return a value describing the error condition:

```
int SP_errno
```

The function `SP_error_message` returns a pointer to the diagnostic message corresponding to a specified error number:

```
char *SP_error_message(int errno)
```

8.2 Calling C from Prolog

Functions written in the C language may be called from Prolog using an interface in which automatic type conversions between Prolog terms and common C types are declared as Prolog facts. Calling without type conversion can also be specified, in which case the arguments and values are passed as SP_term_refs. This interface is partly modeled after Quintus Prolog.

The functions installed using this foreign language interface may invoke Prolog code and use the support functions described in the other sections of this chapter.

Functions, or their equivalent, in any other language having C compatible calling conventions may also be interfaced using this interface. When referring to C functions in the following, we also include such other language functions. Note however that a C compiler is needed since a small amount of glue code (in C) must be generated for interfacing purposes. Note also that on some platforms different calling conventions may exist for C functions. Check the Release Notes to see which one is used by this interface.

8.2.1 Foreign Resource and Conversion Declarations

A *foreign resource* is a set of C functions, defined in one or more files, installed as an atomic operation. The name of a foreign resource, the *resource name*, is an atom, which should uniquely identify the resource. Thus, two foreign resources with the same name cannot be installed at the same time.

For each foreign resource, a `foreign_resource/2` fact is used to declare the interfaced functions. For each of these functions, a `foreign/(2,3)` fact is used to specify conversions between predicate arguments and C-types. These conversion declarations are used for creating the necessary interface between Prolog and C.

The functions making up the foreign resource, the automatically generated interface code, and any libraries, are compiled and linked to form a *linked foreign resource*, which can be statically or dynamically linked to a Development or Runtime System. In all cases, the declared predicates are installed by the built-in predicate `load_foreign_resource/1`. The resource name of a linked foreign resource is derived from its file specification by deleting any leading path and the suffix.

8.2.2 Static and Dynamic linking

The interface can be used in two ways.

With *static linking*, the linked foreign resource is statically linked to a Development or Runtime System. `load_foreign_resource/1` determines if a resource has been statically linked, in which case only the binding of predicate names to functions needs to be done. Otherwise, dynamic linking is attempted.

With *dynamic linking*, the linked foreign resource can be linked and unlinked to a Development or Runtime System, as needed.

A linked foreign resource may be implemented, depending on the hardware and operating system platform, as e.g. a *shared object* or a *dynamic link library*.

8.2.3 Conversion Declarations

Conversion declaration predicates:

`foreign_resource(+`*ResourceName*`,+`*Functions*`)`

A hook predicate. Specifies that a set of foreign functions, to be called from Prolog, are to be found in the resource named by *ResourceName*. *Functions* is a list of functions exported by the resource. Only functions that are to be called from Prolog and optionally one *init function* and one *deinit function* should be listed. The init and deinit functions are specified as `init(`*Function*`)` and `deinit(`*Function*`)` respectively (see Section 8.2.6 [Init and Deinit Functions], page 134). This predicate should be defined entirely in terms of facts (unit clauses). For example:

```
foreign_resource('terminal', [scroll,pos_cursor,ask]).
```

specifies that functions `scroll()`, `pos_cursor()` and `ask()` are to be found in the resource '`terminal`'.

`foreign(+`*CFunctionName*`, +`*Predicate*`)`
`foreign(+`*CFunctionName*`, +`*Language*`, +`*Predicate*`)`

Hook predicates, specify the Prolog interface to a C function. *Language* is at present constrained to the atom `c`. *CFunctionName* is the name of a C function. *Predicate* specifies the name of the Prolog predicate that will be used to call *CFunction()*. *Predicate* also specifies how the predicate arguments are to be translated into the corresponding C arguments. This predicate should be defined entirely in terms of facts (unit clauses). For example:

```
foreign(pos_cursor, c, move_cursor(+integer, +integer)).
```

The above example says that the C function `pos_cursor()` has two integer value arguments and that we will use the predicate `move_cursor/2` to call this function. A goal `move_cursor(5, 23)` would translate into the C call `pos_cursor(5,23);`.

The third argument of the predicate `foreign/3` specifies how to translate between Prolog arguments and C arguments. A call to a foreign predicate will raise an exception if an input arguments is uninstantiated (`instantiation_error/2`) or has the wrong

type (`type_error/4`) or domain (`domain_error/4`). The call will fail upon return from the function if the output arguments do not unify with the actual arguments.

The available conversions are listed in the next subsection.

8.2.4 Conversions between Prolog Arguments and C Types

The following table lists the possible values for the arguments in the predicate specification of `foreign/(2,3)`. The value declares which conversion between corresponding Prolog argument and C type will take place.

`Prolog: +integer`

`C: long` The argument should be a number. It is converted to a C `long` and passed to the C function.

`Prolog: +float`

`C: double` The argument should be a number. It is converted to a C `double` and passed to the C function.

`Prolog: +atom`

`C: unsigned long`
The argument should be an atom. Its canonical representation is passed to the C function.

`Prolog: +chars`

`C: char *` The argument should be a list of character codes. The C function will be passed the address of an array of characters containing these characters. The array is subject to reuse by other support functions, so if the value is going to be used on a more than temporary basis, it must be moved elsewhere.

`Prolog: +string`

`C: char *` The argument should be an atom. The C function will be passed the address of a text string containing the printed representation of the atom. The C function should *not* overwrite the string.

`Prolog: +string(`*N*`)`

`C: char *` The argument should be an atom. The printable representation of the string will be copied into a newly allocated buffer. The string will be truncated if it is longer than *N* characters. The string will be blank padded on the right if it is shorter than *N* characters. The C function will be passed the address of the buffer. The C function may overwrite the buffer, but should not assume that it remains valid after returning.

`Prolog: +address`

`C: void *` The argument should be an integer which value is constrained according to (see Section 8.3.2 [Creating Prolog Terms], page 137, SP_put_address()). The value passed will be a `void *` pointer.

`Prolog: +address(`*TypeName*`)`

`C:` *TypeName* `*`
: The argument should be an integer which value is constrained according to (see Section 8.3.2 [Creating Prolog Terms], page 137, SP_put_address()). The value passed will be a *TypeName* `*` pointer.

`Prolog: +term`

`C: SP_term_ref`
: The argument could be any term. The value passed will be the internal representation of the term.

`Prolog: -integer`

`C: long *` The C function is passed a reference to an uninitialized `long`. The value returned will be converted to a Prolog integer.

`Prolog: -float`

`C: double *`
: The C function is passed a reference to an uninitialized `double`. The value returned will be converted to a Prolog float.

`Prolog: -atom`

`C: unsigned long *`
: The C function is passed a reference to an uninitialized `unsigned long`. The value returned should be the canonical representation of a Prolog atom.

`Prolog: -chars`

`C: char **`
: The C function is passed the address of an uninitialized `char *`. The returned string will be converted to a Prolog list of character codes.

`Prolog: -string`

`C: char **`
: The C function is passed the address of an uninitialized `char *`. The returned string will be converted to a Prolog atom. Prolog will copy the string to a safe place, so the memory occupied by the returned string may be reused during subsequent calls to foreign code.

`Prolog: -string(`*N*`)`

`C: char *` The C function is passed a reference to a character buffer large enough to store an *N* character string. The returned string will be stripped of trailing blanks and converted to a Prolog atom.

`Prolog: -address`

`C: void **`
: The C function is passed the address of an uninitialized `void *`. The returned value, which is constrained according to (see Section 8.3.2 [Creating Prolog Terms], page 137, SP_put_address()), will be converted to a Prolog integer.

`Prolog: -address(`*TypeName*`)`

`C:` *TypeName* `**`
: The C function is passed the address of an uninitialized *TypeName* `*`. The returned value, which is constrained according to (see Section 8.3.2 [Creating Prolog Terms], page 137, SP_put_address()), will be converted to a Prolog integer.

`Prolog: -term`

`C: SP_term_ref`
: The C function is passed a new SP_term_ref, and is expected to set its value to a suitable Prolog term. Prolog will try to unify the value with the actual argument.

`Prolog: [-integer]`

`C: long` *F*`()`
: The C function should return a `long`. The value returned will be converted to a Prolog integer.

`Prolog: [-float]`

`C: double` *F*`()`
: The C function should return a `double`. The value returned will be converted to a Prolog float.

`Prolog: [-atom]`

`C: unsigned long` *F*`()`
: The C function should return an `unsigned long`. The value returned must be the canonical representation of a Prolog atom.

`Prolog: [-chars]`

`C: char *`*F*`()`
: The C function should return a `char *`. The returned string will be converted to a Prolog list of character codes.

`Prolog: [-string]`

`C: char *`*F*`()`
: The C function should return a `char *`. The returned string will be converted to a Prolog atom. Prolog will copy the string to a safe place, so the memory occupied by the returned string may be reused during subsequent calls to foreign code.

`Prolog: [-string(`*N*`)]`

`C: char *`*F*`()`
: The C function should return a `char *`. The first *N* characters of the string will be copied and the copied string will be stripped of trailing blanks. The stripped string will be converted to a Prolog atom. C may reuse or destroy the string buffer during later calls.

`Prolog: [-address]`

`C: void *`*F*`()`
: The C function should return a `void *`. The returned value, which is constrained according to (see Section 8.3.2 [Creating Prolog Terms], page 137, SP_put_address()), will be converted to a Prolog integer.

`Prolog: [-address(`*TypeName*`)]`

`C:` *TypeName* `*`*F*`()`
: The C function should return a *TypeName* `*`. The returned value, which is constrained according to (see Section 8.3.2 [Creating Prolog Terms], page 137, SP_put_address()), will be converted to a Prolog integer.

`Prolog: [-term]`

`C: SP_term_ref` *F*`()`
: The C function should return an SP_term_ref. Prolog will try to unify its value with the actual argument.

8.2.5 Interface Predicates

In the following descriptions, the *Resource* argument is an unsuffixed file specification of a linked foreign resource.

`link_foreign_resource(+`*Resource*`,+`*SourceFile*`,+`*Option*`,+`*CFiles*`,+`*ObjectFiles*`,+`*Libraries*`)`
: Builds a linked foreign resource. *Option* can be either `dynamic` or `static` as described in the alternative form shown below. This predicate is not available in Runtime Systems.

  The work of this predicate is done by a command located in '`$SP_PATH/bin`'. This command is configurable as described in the Release Notes. It can also be invoked directly from a shell as an alternative to running it from Prolog:

`splfr` *Resource SourceFile* `[+dynamic | +static] [+c` *CFiles*`] [+o` *ObjectFiles*`] [+l` *Libraries*`] [+copt` *COptions*`]`
: Reads terms from *SourceFile*, applying term expansion and extracting any `foreign_resource/2` fact with first argument matching the name of *Resource* and all `foreign/(2,3)` facts. Compiles *CFiles* and the interface code derived from these facts and links the resulting object files with *ObjectFiles* and *Libraries* to form the linked foreign resource *Resource*. The option `+dynamic` (`+static`) specifies that the linked foreign resource is suitable for dynamic (static) linking. The default is dynamic. The option `+copt` can be used to pass options to the C-compiler.

`load_foreign_resource(:`*Resource*`)`
: Unless a foreign resource with the same name as *Resource* has been statically linked, the linked foreign resource specified by *Resource* is linked into the Prolog load image. In both cases, the predicates defined by *Resource* are installed, and any init function is called. Dynamic linking is not possible if the foreign resource was linked using the `+static` option.

  If a resource with the same name has been previously loaded, it will be unloaded as if `unload_foreign_resource(`*Resource*`)` was called, before *Resource* is loaded.

  In Muse, the system is required to be adjusted to one worker first, and the predicates defined become *cavalier* (see Section 3.5 [Programming Considerations], page 45).

`unload_foreign_resource(:`*Resource*`)`
: Any deinit function associated with *Resource* is called, and the predicates defined by *Resource* are uninstalled. If *Resource* has been dynamically linked, it is unlinked from the Prolog load image.

In Muse, the system is required to be adjusted to one worker first, and the predicates defined become *cavalier*

The following predicates are provided for backwards compatibility and should be avoided in new code:

`foreign_file(+`*File*`,+`*Functions*`)`
: *A hook predicate.* Specifies that a set of foreign functions, to be called from Prolog, are to be found in *File*. This predicate is only called from `load_foreign_files/2`.

`load_foreign_files(:`*ObjectFiles*`,+`*Libraries*`)`
: *Hookable*, a resource name is derived from the first file name in *ObjectFiles* by stripping off the suffix. If this resource has been statically linked, the predicates defined by it are installed; otherwise, a linked foreign resource containing the declared functions is created and loaded. Not available in Runtime Systems.

8.2.6 Init and Deinit Functions

An *init function* and/or a *deinit function* can be declared by `foreign_resource/2`. If this is the case, these functions should have the prototype:

`void` *FunctionName* `(int when)`

The init function is called by `load_foreign_resource/1` after the resource has been loaded and the interfaced predicates have been installed.

The deinit function is called by `unload_foreign_resource/1` before the interfaced predicates have been uninstalled and the resource has been unloaded.

The init and deinit functions may use the C-interface to call Prolog etc.

Foreign resources are unloaded before saving states, and reloaded afterwards or when the saved state is restored; see Section 1.10 [Saving], page 17. Foreign resources are also unloaded when exiting Prolog execution. The parameter `when` reflects the context of the `(un)load_foreign_resource/1` and is set as follows for init functions:

`SP_WHEN_EXPLICIT`
: Explicit call to `load_foreign_resource/1`.

`SP_WHEN_RESTORE`
: Resource is reloaded after save or restore.

For deinit functions:

`SP_WHEN_EXPLICIT`
: Explicit call to `unload_foreign_resource/1`.

`SP_WHEN_SAVE`
: Resource is unloaded before save.

`SP_WHEN_EXIT`
: Resource is unloaded before exiting Prolog.

8.2.7 Creating the Linked Foreign Resource

Suppose we have a Prolog source file `ex.pl` containing:

```
foreign(f1, p1(+integer,[-integer])).
foreign(f2, p2(+integer,[-integer])).
foreign_resource(ex, [f1,f2]).
:- load_foreign_resource(ex).
```

and a C source file `ex.c` with definitions of the functions `f1` and `f2`, both returning `long` and having a `long` as only parameter. The conversion declarations in '`ex.pl`' state that these functions form the foreign resource `ex`.

To create the linked foreign resource, simply type (to Prolog):

```
| ?- link_foreign_resource(ex,'ex.pl',dynamic,['ex.c'],[],[]).
```

or alternatively (to the Shell):

```
% splfr ex ex.pl +c ex.c
```

The linked foreign resource '`ex.so`' (file suffix '`.so`' is system dependent) has been created. It will be dynamically linked by the directive `:- load_foreign_resource(ex).` when the file '`ex.pl`' is loaded. Linked foreign resources can also be created manually (see Chapter 38 [Runtime Utilities], page 373).

Dynamic linking of foreign resources can also be used by Runtime Systems. On some platforms, however, the executable must not be *stripped* for dynamic linking to work, i.e. its symbol table must remain.

8.3 Support Functions

The support functions include functions to manipulate SP_term_refs, functions to convert data between the basic C types and Prolog terms, functions to test whether a term can be converted to a specific C type, and functions to unify or compare two terms.

8.3.1 Creating and Manipulating SP_term_refs

Normally, C functions only have indirect access to Prolog terms via SP_term_refs. C functions may receive arguments as unconverted Prolog terms, in which case the actual arguments received will have the type `SP_term_ref`. Also, a C function may return an unconverted Prolog term, in which case it must create an SP_term_ref. Finally, any temporary Prolog terms created by C code must be handled as SP_term_refs.

SP_term_refs are motivated by the fact that SICStus Prolog's memory manager must have a means of reaching all live Prolog terms for memory management purposes, including such terms that are being manipulated by the user's C code. Previous releases of SICStus Prolog provided direct access to Prolog terms and the ability to tell the memory manager that a given memory address points to a Prolog term, but this approach was too low level and highly error-prone. The current design is modeled after and largely compatible with Quintus Prolog release 3.

SP_term_refs are created dynamically. At any given time, an SP_term_ref has a value (a Prolog term). This value can be examined, accessed, and updated by the support functions described in this section.

It is important to understand the rules governing the scope of SP_term_refs in conjunction with calls from Prolog to C and vice versa:

- When a C function called from Prolog returns, all SP_term_refs passed to the function or dynamically created by the function become invalid.
- When terms are passed to C as a result of calling Prolog, those terms and any SP_term_refs created since the start of the query are only valid up to the request for the next solution. They also become invalid when the query is closed.

A new SP_term_ref whose value is `[]` is created by calling

```
SP_term_ref SP_new_term_ref(void)
```

The value of the SP_term_ref `to` is set to the value of the SP_term_ref `from` by calling `SP_put_term(to,from)`. The previous value of `to` is lost:

```
void SP_put_term(SP_term_ref to, SP_term_ref from)
```

Each Prolog atom is represented internally by a unique integer, represented in C as an `unsigned long`. This mapping between atoms and integers depends on the execution history. Certain functions require this representation as opposed to an SP_pred_ref. It can be obtain by a special argument type declaration when calling C from Prolog, by calling `SP_get_atom()`, or by looking up a string `s` in the Prolog symbol table by calling `SP_atom_from_string(s)`:

```
unsigned long SP_atom_from_string(char *s)
```

The print name of a Prolog atom `a` can be obtained by calling:

```
char *SP_string_from_atom(unsigned long a)
```

The print name length of a Prolog atom `a` can be obtained by calling:

```
int SP_atom_length(unsigned long a)
```

Same as `strlen(SP_string_from_atom(a))` but runs in O(1) time.

Prolog atoms, and the space occupied by their print names, are subject to garbage collection when the number of atoms has reached a certain threshold, under the control of the `agc_margin` Prolog flag (see Section 7.6 [State Info], page 104), or when the atom garbage collector is called explicitly. The atom garbage collector will find all references to atoms from the Prolog specific memory areas, including SP_term_refs and arguments passed from Prolog to foreign language functions. However, atoms created by `SP_atom_from_string` and merely stored in a local variable are endangered by garbage collection. The following functions make it possible to protect an atom while it is in use. The operations are implemented using reference counters to cater for multiple, independent use of the same atom in different foreign resources:

```
int SP_register_atom(unsigned long a)
```

Registers the atom `a` with the Prolog memory manager by incrementing its reference counter. Returns a nonzero value if the operation succeeds.

```
int SP_unregister_atom(unsigned long a)
```

Unregisters the atom `a` with the Prolog memory manager by decrementing its reference counter. Returns a nonzero value if the operation succeeds.

8.3.2 Creating Prolog Terms

These functions create a term and store it as the value of an SP_term_ref, which must exist prior to the call. They return zero if the conversion fails (as far as failure can be detected), and a nonzero value otherwise, assigning to `t` the converted value.

`int SP_put_variable(SP_term_ref t)`
: Assigns to `t` a new Prolog variable.

`int SP_put_integer(SP_term_ref t, long l)`
: Assigns to `t` a Prolog integer from a C long integer.

`int SP_put_float(SP_term_ref t, double d)`
: Assigns to `t` a Prolog float from a C double.

`int SP_put_atom(SP_term_ref t, unsigned long a)`
: Assigns to `t` a Prolog atom from `a`, which must be the canonical representation of a Prolog atom. (see Section 8.2 [Calling C], page 128).

`int SP_put_string(SP_term_ref t, char *name)`
: Assigns to `t` a Prolog atom from a C string.

`int SP_put_address(SP_term_ref t, void *pointer)`
: Assigns to `t` a Prolog integer from a C pointer.

  The pointer must be `NULL` or an address having the four most significant bits in common with addresses returned by `malloc()`. (This excludes pointers to automatic C variables,

i.e. addresses to the C stack). Furthermore, the address must be aligned on a four bytes boundary.

`NULL` is converted to the integer `0`.

`int SP_put_list_chars(SP_term_ref t, SP_term_ref tail, char *s)`
: Assigns to `t` a Prolog list of the character codes in `s`. The list is terminated by the value of `tail`.

`int SP_put_list_n_chars(SP_term_ref t, SP_term_ref tail, long n, char *s)`
: Assigns to `t` a Prolog list of the first `n` character codes or zeroes in `s`. The list is terminated by the value of `tail`.

`int SP_put_number_chars(SP_term_ref t, char *s)`
: Assigns to `t` a Prolog number by parsing the string in `s`.

`int SP_put_functor(SP_term_ref t, unsigned long name, int arity)`
: Assigns to `t` a Prolog compound term with all the arguments unbound variables. If `arity` is 0, assigns the Prolog atom whose canonical representation is `name` to `t`. This is similar to calling `functor/3` with the first argument unbound and the second and third arguments bound to an atom and an integer, respectively.

`int SP_put_list(SP_term_ref t)`
: Assigns to `t` a Prolog list whose head and tail are both unbound variables.

`int SP_cons_functor(SP_term_ref t, unsigned long name, int arity, SP_term_ref arg, ...)`
: Assigns to `t` a Prolog compound term whose arguments are the values of `arg`... If `arity` is 0, assigns the Prolog atom whose canonical representation is `name` to `t`. This is similar to calling `=../2` with the first argument unbound and the second argument bound.

`int SP_cons_list(SP_term_ref t, SP_term_ref head, SP_term_ref tail)`
: Assigns to `t` a Prolog list whose head and tail are the values of `head` and `tail`.

8.3.3 Accessing Prolog Terms

These functions will take an SP_term_ref and convert it to C data. They return zero if the conversion fails, and a nonzero value otherwise, and (except the last one) store the C data in output arguments.

`int SP_get_integer(SP_term_ref t, long *l)`
: Assigns to `*l` the C `long` corresponding to a Prolog number. The value must fit in `*l` for the operation to succeed.

`int SP_get_float(SP_term_ref t, double *d)`
: Assigns to `*d` the C `double` corresponding to a Prolog number.

`int SP_get_atom(SP_term_ref t, unsigned long *a)`
: Assigns to `*a` the canonical representation of a Prolog atom.

`int SP_get_string(SP_term_ref t, char **name)`
: Assigns to `*name` a pointer to the string that is the name of a Prolog atom. This string must *not* be modified.

`int SP_get_address(SP_term_ref t, void **pointer)`
: Assigns to `*pointer` a C pointer from a Prolog term. The term should be an integer which value is constrained according to (see Section 8.3.2 [Creating Prolog Terms], page 137, SP_put_address()).

`int SP_get_list_chars(SP_term_ref t, char **s)`
: Assigns to `*s` a zero-terminated array of characters corresponding to a Prolog list of character codes. The array is subject to reuse by other support functions, so if the value is going to be used on a more than temporary basis, it must be moved elsewhere.

`int SP_get_list_n_chars(SP_term_ref t, SP_term_ref tail, long n, long *w, char *s)`
: Copies into `s` at most `n` character codes or zeroes from the list `t`. The number of character codes actually written is assigned to `*w`. `tail` is set to the remainder of the list. The array `s` must have room for at least `n` characters.

`int SP_get_number_chars(SP_term_ref t, char **s)`
: Assigns to `*s` a zero-terminated array of characters corresponding to the printed representation of a Prolog number. The array is subject to reuse by other support functions, so if the value is going to be used on a more than temporary basis, it must be moved elsewhere.

`int SP_get_functor(SP_term_ref t, unsigned long *name, int *arity)`
: Assigns to `*name` and `*arity` the canonical representation and arity of the principal functor of a Prolog compound term. If the value of `t` is an atom, then that atom is assigned to `*name` and 0 is assigned to `*arity`. This is similar to calling `functor/3` with the first argument bound to a compound term or an atom and the second and third arguments unbound.

`int SP_get_list(SP_term_ref t, SP_term_ref head, SP_term_ref tail)`
: Assigns to `head` and `tail` the head and tail of a Prolog list.

`int SP_get_arg(int i, SP_term_ref t, SP_term_ref arg)`
: Assigns to `arg` the `i`:th argument of a Prolog compound term. This is similar to calling `arg/3` with the third argument unbound.

8.3.4 Testing Prolog Terms

There is one general function for type testing of Prolog terms and a set of specialized, more efficient, functions, one for each term type.

`int SP_term_type(SP_term_ref t)`
: Depending on the type of the term t, one of `SP_TYPE_VARIABLE`, `SP_TYPE_INTEGER`, `SP_TYPE_FLOAT`, `SP_TYPE_ATOM`, or `SP_TYPE_COMPOUND` is returned.

`int SP_is_variable(SP_term_ref t)`
: Returns nonzero if the term is a Prolog variable, zero otherwise.

`int SP_is_integer(SP_term_ref t)`
: Returns nonzero if the term is a Prolog integer, zero otherwise.

`int SP_is_float(SP_term_ref t)`
: Returns nonzero if the term is a Prolog float, zero otherwise.

`int SP_is_atom(SP_term_ref t)`
: Returns nonzero if the term is a Prolog atom, zero otherwise.

`int SP_is_compound(SP_term_ref t)`
: Returns nonzero if the term is a Prolog compound term, zero otherwise.

`int SP_is_list(SP_term_ref t)`
: Returns nonzero if the term is a Prolog list, zero otherwise.

`int SP_is_atomic(SP_term_ref t)`
: Returns nonzero if the term is an atomic Prolog term, zero otherwise.

`int SP_is_number(SP_term_ref t)`
: Returns nonzero if the term is a Prolog number, zero otherwise.

8.3.5 Unifying and Comparing Terms

`int SP_unify(SP_term_ref x, SP_term_ref y)`
: Unifies two terms, returning zero on failure and nonzero on success.

`int SP_compare(SP_term_ref x, SP_term_ref y)`
: Returns `-1` if `x @< y`, `0` if `x == y` and `1` if `x @> y`

8.3.6 Memory Allocation

The usual C library memory allocation functions (`malloc`, `realloc`, and `free`) may not work properly in foreign code. The following functions provide these services from SICStus Prolog's memory manager:

`void *SP_malloc(unsigned int size)`
: Returns a properly aligned pointer to a block of at least `size` bytes. In Muse, the blocks are allocated in shared memory.

`void *SP_realloc(void *ptr, unsigned int size)`
: Changes the size of the block referenced by `ptr` to `size` bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes. The block referenced by `ptr` must have been obtained by a call to `SP_malloc` or `SP_realloc`, and must not have been released by a call to `SP_free` or `SP_realloc`. In Muse, the blocks are allocated in shared memory.

`void SP_free(void *ptr)`
: Releases the block referenced by `ptr`, which must have been obtained by a call to `SP_malloc` or `SP_realloc`, and must not have been released by a call to `SP_free` or `SP_realloc`. In Muse, the blocks are allocated in shared memory.

8.4 Calling Prolog from C

In the sequential Development System and in Runtime Systems, Prolog and C code may call each other to arbitrary depths. The ability to call Prolog from C is not available in the Muse Development System in this release.

Before calling a predicate from C you must look up the predicate definition by module, name, and arity. The function `SP_predicate()` will return a pointer to this definition or return `NULL` if the predicate is not visible in the module. This definition could be used in more than one call to the same predicate. The module specification is optional. If `NULL` or `""` (the empty string) is given then the default *type-in* module (see Section 4.2 [Module Spec], page 48) is assumed:

```
SP_pred_ref SP_predicate(char *name_string,
                         long arity,
                         char *module_string)
```

The function `SP_pred()` may be used as an alternative to the above. The only difference is that the name and module arguments are passed as Prolog atoms rather than strings, and the module argument is mandatory. This saves the cost of looking up the two arguments in the Prolog symbol table. This cost dominates the cost of `SP_predicate()`.

```
SP_pred_ref SP_pred(unsigned long name_atom,
                    long arity,
                    unsigned long module_atom)
```

8.4.1 Finding One Solution of a Call

The easiest way to call a predicate if you are only interested in the first solution is to call the function `SP_query()`. It will create a goal from the predicate definition and the arguments, and try to prove it. The call will return `SP_SUCCESS` if the goal succeeded, `SP_FAILURE` if it failed, and `SP_ERROR` if an error condition occurred.

```
int SP_query(SP_pred_ref predicate, SP_term_ref arg1, ...)
```

If you are only interested in the side effects of a predicate you can call `SP_query_cut_fail()`. It will try to prove the predicate, cut away the rest of the solutions, and finally fail. This will reclaim the storage used after the call:

```
int SP_query_cut_fail(SP_pred_ref predicate, SP_term_ref arg1, ...)
```

8.4.2 Finding Multiple Solutions of a Call

If you are interested in more than one solution a more complicated scheme is used. You find the predicate definition as above but you don't call the predicate directly.

1. Set up a call with `SP_open_query()`

2. Call `SP_next_solution()` to find a solution. Call this predicate again to find more solutions if there are any.
3. Terminate the call with `SP_close_query()` or `SP_cut_query()`

The function `SP_open_query()` will return an identifier of type `SP_qid` that you use in successive calls, or `NULL`, if given an invalid predicate reference. Note that if a new query is opened while another is already open, the new query must be terminated before performing any action on the old one. That is, queries must be strictly nested:

```
SP_qid SP_open_query(SP_pred_ref predicate, SP_term_ref arg1, ...)
```

The function `SP_next_solution()` will cause the Prolog engine to find solutions of the open query. The SP_term_refs that you sent with the call to `SP_open_query()` will be assigned new values. `SP_next_solution()` will return `SP_SUCCESS` for success, `SP_FAILURE` for failure, `SP_ERROR` if an error condition occurred.

```
int SP_next_solution(SP_qid query)
```

You can terminate a query in two ways. The function `SP_cut_query()` will only take away the choices created since the corresponding `SP_open_query()`. The data created in the call are still valid and could be passed to another call. The function `SP_close_query()` will restore the state to what it was before the call to `SP_open_query()`. If the call created data that you want to keep, it must be converted to C data before calling `SP_close_query()`. Both functions return `SP_SUCCESS` for success and `SP_ERROR` for invalid usage:

```
int SP_cut_query(SP_qid query)

int SP_close_query(SP_qid query)
```

8.4.3 Calling Prolog Asynchronously

A Prolog execution may be interrupted by signals or similar asynchronous events. If you wish to call Prolog back from a signal handler you cannot use `SP_query()` etc. directly. The call to Prolog has to be delayed until a time when the Prolog execution can accept an interrupt. The function `SP_event()` serves this purpose, and installs the function `func` to be called from Prolog when the execution can accept a callback. Returns non-zero iff installation succeeded. `func` is called with `arg` as first argument.

A queue of functions, with corresponding arguments, is maintained; that is, if several calls to `SP_event()` occur before Prolog can a accept an interrupt, the functions are queued and executed in turn at the next possible opportunity. Note that the queuing facility is only safe for signal handlers installed using `SP_signal()` (see below).

Depending on the value returned from `func`, the interrupted Prolog execution will just continue (`SP_SUCCESS`) or backtrack (`SP_FAILURE` or `SP_ERROR`). An exception raised by `func` will be processed in the interrupted Prolog execution. In case of fail or exception the event queue is flushed:

```
int SP_event(int (*func)(), void *arg)
```

A signal handler having called `SP_event()` should call `SP_continue()` as its last action, to ensure that the interrupt is processed as early as possible:

```
void SP_continue()
```

To install a function, `func`, as a handler for the signal `sig`, call:

```
void (*SP_signal (int sig, void (*func)()))()
```

`SP_signal()` will also, if permitted by the operating system, add `sig` to a set of signals which are all blocked during the handling of the event queue. Some operating systems require that:

```
void (*SP_reinstall_signal (int sig, void (*func)()))()
```

be called from a signal handler to unblock or reinstall the handler. This function should be called *before* `SP_continue()`.

The following piece of C code illustrates these facilities. The function `signal_init()` installs the function `signal_handler()` as the primary signal handler for the signals `USR1` and `USR2`. That function invokes the predicate `prolog_handler/1` as the actual signal handler, passing the signal number as an argument to the predicate.

```
SP_pred_ref event_pred;

static int signal_event(signal_no)
     void *signal_no;
{
  SP_term_ref x=SP_new_term_ref();
  int rc;

  SP_put_integer(x, (int)signal_no);
  rc = SP_query(event_pred, x);
  if (rc == SP_ERROR && SP_exception_term(x))
    SP_raise_exception(x);        /* Propagate any raised exception */
  return rc;
}

static void signal_handler(sig)
     int sig;
{
  SP_event(signal_event, (void *)sig);
  SP_reinstall_signal(sig, signal_handler);
  SP_continue();
}

void signal_init()
{
  event_pred = SP_predicate("prolog_handler",1,"");
```

```
  SP_signal(SIGUSR1, signal_handler);
  SP_signal(SIGUSR2, signal_handler);
}
```

8.4.4 Exception Handling in C

When an exception has been raised, the functions `SP_query()`, `SP_query_cut_fail()` and `SP_next_solution()` return `SP_ERROR`. To access the *exception term* (the argument of the call to `raise_exception/1`), which is asserted when the exception is raised, the function `SP_exception_term()` is used. As a side effect, the exception term is retracted, so if your code wants to pass the exception term back to Prolog, it must use the `SP_raise_exception()` function below. If an exception term exists, `SP_exception_term()` retracts it and stores it as the value of an SP_term_ref which must exist prior to the call and returns nonzero. Otherwise, it returns zero.

```
int SP_exception_term(SP_term_ref t)
```

To raise an exception from a C function called from Prolog, just call `SP_raise_exception(t)` where `t` is the SP_term_ref whose value is the exception term. The glue code will detect that an exception has been raised, any value returned from the function will be ignored, and the exception will be passed back to Prolog.

```
void SP_raise_exception(SP_term_ref t)
```

8.5 SICStus Streams

With the SICStus Prolog C interface, the user can define his/her own streams as well as from C read or write on the predefined streams. The stream interface is modeled after Quintus Prolog release 2. It provides:

- C functions to perform I/O on Prolog streams. This way you can use the same stream from Prolog and C code.
- User defined streams. You can define your own Prolog streams in C.
- Bidirectional streams. A SICStus stream supports reading or writing or both.
- Hookable standard input/output/error streams.

8.5.1 Prolog Streams

From the Prolog level there is a unique number that identifies a stream. This identifier can be converted from/to a Prolog stream:

`stream_code(+`*Stream*`,` *?StreamCode*`)`
`stream_code(`*?Stream*`,+`*StreamCode*`)`
: *StreamCode* is the C stream identifier (an integer) corresponding to the Prolog stream *Stream*. This predicate is only useful when streams are passed between Prolog and C. Note that *StreamCode* no longer has any relation to the file descriptor.

The *StreamCode* is a Prolog integer representing a `SP_stream *` pointer as described in (see Section 8.3.2 [Creating Prolog Terms], page 137, SP_put_address()).

To write on a Prolog stream from C, special versions of the most common standard C I/O functions are used:

```
int SP_getc(void)
int SP_fgetc(SP_stream *s)
void SP_putc(int c)
void SP_fputc(int c, SP_stream *s)
void SP_puts(char *string)
void SP_fputs(char *string, SP_stream *s)
int SP_printf(char *format, ...)
int SP_fprintf(SP_stream *s, char *format, ...)
int SP_fflush(SP_stream *s)
int SP_fclose(SP_stream *s)
```

There are three predefined streams accessible from C:

`SP_stdin` Standard input. Refers to the same stream as `user_input` in Prolog. Which stream is referenced by `user_input` is controlled by the flag `user_input` (see `prolog_flag/3`) .

`SP_stdout`
: Standard output. Refers to the same stream as `user_output` in Prolog. Which stream is referenced by `user_output` is controlled by the flag `user_output` (see `prolog_flag/3`).

`SP_stderr`
: Standard error. Refers to the same stream as `user_error` in Prolog. Which stream is referenced by `user_error` is controlled by the flag `user_error` (see `prolog_flag/3`).

`SP_curin` Current input. It is initially set equal to `SP_stdin`. It can be changed with the predicates `see/1` and `set_input/1`.

`SP_curout`
: Current output. It is initially set equal to `SP_stdout`. It can be changed with the predicates `tell/1` and `set_output/1`.

Note that these variables are read only. They are set but never read by the stream handling.

8.5.2 Defining a New Stream

The following steps are required to define a new stream in C:

- Define low level functions (character reading, writing etc).
- Initialize and open your stream.
- Allocate memory needed for your particular stream.
- Initialize and install a Prolog stream with `SP_make_stream()`.
- Initialize additional fields. Some streams may require additional changes to the fields in the `SP_stream` structure than the default values set by `SP_make_stream()`.

8.5.2.1 Low Level I/O Functions

For each new stream the appropriate low level I/O functions have to be defined. Error handling, prompt handling and character counting is handled in a layer above these functions. They all operate on a user defined private data structure pointed out by `user_handle` in `SP_stream`.

User defined low level I/O functions may invoke Prolog code and use the support functions described in the other sections of this chapter.

`int` *my˙fgetc*`(void *handle)`
: Should return the character read or -1 on end of file.

`int` *my˙fputc*`(char c, int handle)`
: Should write the character `c` and return the character written.

`int` *my˙flush*`(void *handle)`
: Flush the stream. Should return 0 on success, EOF on error.

`int` *my˙eof*`(void *handle)`
: Should return 1 on end of file, else 0.

`void` *my˙clrerr*`(void *handle)`
: Clear the error level.

`int` *my˙close*`(void *handle)`
: Close the stream. Should return zero.

8.5.2.2 Installing a New Stream

A new stream is made accessible to Prolog with the function `SP_make_stream()`.

```
int SP_make_stream(
        void *handle,
        int (*sgetc)(),
        int (*sputc)(),
        int (*sflush)(),
        int (*seof)(),
        void (*sclrerr)(),
        int (*sclose)(),
        SP_stream **stream)
```

The function will:

- Allocate a `SP_stream` structure
- Install your low level I/O functions. For those not supplied default functions are installed.
- Determine if the stream is for input or output or both from the functions supplied.
- Fill in fields in the `SP_stream` structure with default values

The `handle` pointer will be supplied as the `handle` argument in the calls to the low level functions.

A stream without a close function will be treated as not closable i.e. `close/1` will not have any effect on it.

8.5.2.3 Internal Representation

For most streams you don't have to know anything about the internal representation but there may be occasions when you have to set some fields manually or do some processing on all streams of a particular type. SICStus Prolog maintain a circular list of stream objects of type `SP_stream`.

`SP_stream *backward;`
`SP_stream *forward;`
: Used for linking streams together. The insertion is done by `SP_make_stream()` and the deletion is done from the Prolog predicate `close/1`.

`char *filename;`
: This field is set to the empty string, `""`, by `SP_make_stream()`. May be set to a suitable string, provided the string will not be overwritten until the stream is closed.

`unsigned long mode;`
: A bit vector that contains information about the access modes supported, if the stream is a TTY stream etc. It is not available to the user but the TTY property can be set by the function:

  `void SP_set_tty(SP_stream *s)`

`int fd;` The I/O descriptor if the stream is associated with a file, socket etc. Otherwise a negative number.

`void *user_handle;`
This is the pointer to the user supplied private data for the stream. In the case of SICStus Prolog predefined file streams the `user_handle` could be a pointer to the standard I/O `FILE`.

There is no standard way to tell if a stream is user defined. You have to save pointers to the streams created or check if one of the stream functions installed is user defined, i.e:

```
int is_my_stream(SP_stream *s)
{
  return (s->sclose == my_close);
}
```

8.5.3 Hookable Standard Streams

As of SICStus release 3.7, the standard I/O streams (input, output, and error) are hookable, i.e. the streams can be redefined by the user.

`SP_UserStreamHook *SP_set_user_stream_hook(SP_UserStreamHook *hook)`
Sets the user-stream hook to `hook`.

`SP_UserStreamPostHook *SP_set_user_stream_post_hook(SP_UserStreamPostHook *hook)`
Sets the user-stream post-hook to `hook`.

8.5.3.1 Writing User-stream Hooks

The user-stream hook is, if defined, called during `SP_initialize()`. `SP_set_user_stream_hook()` and `SP_set_user_stream_post_hook()` must therefore be called before `SP_initialize()` is called. It has the following prototype:

```
SP_stream *user_stream_hook(int which)
```

If the hook is not defined, SICStus will attempt to open the standard TTY/console versions of these streams. If they are unavailable (such as for windowed applications under Windows), the result is undefined.

It is called three times, one for each stream. The `which` argument indicates which stream it is called for. The value of `which` is one of:

`SP_STREAMHOOK_STDIN`
Create stream for standard input.

`SP_STREAMHOOK_STDOUT`
Create stream for standard output.

`SP_STREAMHOOK_STDERR`
: Create stream for standard error.

The hook should return a standard SICStus I/O stream, as described in Section 8.5.2 [Defining a New Stream], page 146.

8.5.3.2 Writing User-stream Post-hooks

The user-stream post-hook is, if defined, called after all the streams have been defined, once for each of the three standard streams. It has a slightly different prototype:

```
void user_stream_post_hook(int which, SP_stream *str)
```

where `str` is a pointer to the corresponding `SP_stream` structure. There are no requirements as to what this hook must do; the default behavior is to do nothing at all.

The post-hook is intended to be used to do things which may require that all streams have been created.

8.5.3.3 User-stream Hook Example

This section contains an example of how to create and install a set of user-streams.

The hook is set by calling `SP_set_user_stream_hook()` in the main program like this:

```
SP_set_user_stream_hook((SP_UserStreamHook *)user_strhook);
```

Remember: `SP_set_user_stream_hook()` and `SP_set_user_stream_post_hook()` must be called *before* `SP_initialize()`.

The hook `user_strhook()` is defined like this:

```
SP_stream *user_strhook(int std_fd)
{
  SP_stream *s;

  SP_make_stream(NULL, my_getc, my_putc, my_flush, my_eof, my_clrerr, NULL, &s);

  return s;
}
```

See Section 8.5.2.2 [Installing a New Stream], page 147 for a description on the parameters to `SP_make_stream()`.

8.6 Muse Support Functions

Dynamic loading of foreign language functions must be done when the system is adjusted to one worker. If you use dynamic linking, it is not recommended to use C-global or static local variables. It is platform dependent if the variables are shared among workers or private. If static linking is used, the variables become private.

Some useful Muse C-functions:

`int muse_max_workers(void)`
: Returns the maximum number of workers. Also available as a read-only Muse flag.

`int muse_num_workers(void)`
: Returns the actual number of allocated workers. Also available as a writable Muse flag.

`int muse_worker_id(void)`
: Returns the worker identity as a non-negative integer. Also available as a read-only Muse flag.

`void muse_init_lock(int *lock)`
`void muse_lock(int *lock)`
`void muse_un_lock(int *lock)`
: The `muse_lock` and `muse_un_lock` functions is used to implement mutual exclusion regions. **Note:** the actual locks in Muse may not be `int` but they are of equal or smaller size. A Muse lock must be initialized with `muse_init_lock()` before it can be used.

In Muse, dynamic memory allocation is performed in shared memory. To allocate worker private memory, it is recommended (as `worker_counters` in Section 10.5.9 [Muse FLI Example], page 184) to allocate an array of size `muse_max_workers()` that is indexed by the worker id (`muse_worker_id()`).

8.7 Hooks

The user may define functions to be called at certain occasions by the Prolog system. This is accomplished by passing the functions as arguments to the following set-hook-functions. The functions can be removed by passing a `NULL`.

```
typedef int (SP_ReadHookProc) (int fd)
SP_ReadHookProc SP_set_read_hook (SP_ReadHookProc *)
```
: The installed function is called before reading a character from `fd` provided it is associated with a terminal device. This function shall return nonzero when there is input available at `fd`. It is called repeatedly until it returns nonzero. Not available in Muse.

```
typedef void (SP_VoidFun) (void)
SP_VoidFun * SP_set_reinit_hook (SP_VoidFun *)
```
: The installed function is called upon abort and reinitialization. The call is made after SICStus Prolog's signal handler installation but before the built-in predicates `version/0` and `initialization/0` are called. Calling Prolog from functions invoked through this hook is not supported. (This hook is not available in Runtime Systems.)

```
typedef void (SP_VoidFun) (void);
```

`SP_VoidFun * SP_set_interrupt_hook (SP_VoidFun *)`

The installed function is called on occasions like expansion of stacks, garbage collection and printouts, in order to yield control to special consoles etc. for interrupt checking. Calling Prolog from functions invoked through this hook is not supported. (This hook is not available in Runtime Systems.)

8.8 Runtime Systems

It is possible to mix C and Prolog code to create stand-alone applications, *Runtime Systems*. In a Runtime System there are some differences in the behavior of the Prolog engine and many built-in predicates are omitted or have limited functionality:

- No top level
- No debugger
- No compiler
- No progress or error messages while loading files
- No profiling
- No OR-parallelism
- No redefinition warnings
- No signal handling except as installed by `SP_signal()`
- Uncaught exceptions are returned to C
- The predicates `halt/0`, `abort/0` and `reinitialise/0` will return to C
- You have to supply a main program and explicitly initialize the Prolog engine with `SP_initialize()`.

8.8.1 Initializing the Prolog Engine

You must call `SP_initialize()` before calling any other interface function. The function will allocate data areas used by Prolog, initialize command line arguments so that they can be accessed by the `argv` Prolog flag, and load the Runtime Library:

```
int SP_initialize(int argc, char **argv, char *boot_path)
```

`boot_path` should be the name of a directory, equivalent to '`$SP_PATH/bin`'. If `boot_path` is `NULL`, `SP_initialize()` will look up the value of the environment variable `SP_PATH` and look for the file '`$SP_PATH/bin/sprt.sav`' which contains the Runtime Library.

It returns `SP_SUCCESS` if initialization was successful, and `SP_ERROR` otherwise.

Optionally, you may also call `SP_force_interactive()` before calling `SP_initialize()`. This will force the I/O built-in predicates to treat the standard input stream as a terminal, even if it does not appear to be a terminal. Same as the '`-i`' option in Development Systems. (see Section 1.1 [Start], page 7).

```
void SP_force_interactive(void)
```

Optionally, you may also call `SP_set_memalloc_hooks()` before calling `SP_initialize()`. This will define the bottom layer of Prolog's memory manager, in case your application has special requirements.

```
typedef void *(SP_AllocHook)(unsigned int size,
                             unsigned int align,
                             unsigned int *actual_sizep);
typedef void *(SP_ReAllocHook)(void *ptr,
                               unsigned int oldsize,
                               unsigned int newsize,
                               unsigned int align,
                               unsigned int *actual_sizep);
typedef int (SP_FreeHook)(void *ptr,
                          unsigned int size);

void SP_set_memalloc_hooks(int usage,
                           SP_AllocHook *init_alloc_hook,
                           SP_AllocHook *alloc_hook,
                           SP_ReAllocHook *realloc_hook,
                           SP_FreeHook *free_hook)
```

The effect of `SP_set_memalloc_hooks` is controlled by the value of `usage`, which should be one of:

`MM_USE_MALLOC`
: The bottom layer will be based on `malloc()`/`free()`. The other arguments are ignored. Same as the '`-m`' option in Development Systems. (see Section 1.1 [Start], page 7).

`MM_USE_SBRK`
: The bottom layer will be based on `sbrk()`. The default for UNIX; not available for Windows. The other arguments are ignored.

`MM_USE_SPARSE`
: The bottom layer will be based on `VirtualAlloc()`/`VirtualFree()`. The default for Windows; not available for UNIX. The other arguments are ignored.

`MM_USE_OTHER`
: The bottom layer will be based on the other arguments. Their meaning is explained below.

In the latter case, the other arguments should be functions as specified below:

`alloc_hook`
: must allocate and return a pointer to a piece of memory that has at least `size` bytes aligned at `align` in it. `align` is guaranteed to be a power of 2. The actual size of the piece of memory should be returned in `*actual_sizep`. Should return `NULL` if it cannot allocate any more memory.

`init_alloc_hook`
: is a special case of `alloc_hook`. It will be called initially whereas `alloc_hook` will be called subsequently. It can do whatever initialization that this layer of memory management wants to do.

`realloc_hook`
: is called with a piece of memory to be resized and possibly moved. `ptr` is the pointer, `oldsize` its current size. The function must allocate and return a pointer to a piece of memory that has at least `newsize` bytes aligned at `align` in it, and that has the same contents as the old block up to the lesser of `oldsize` and `newsize`. `align` is guaranteed to be a power of 2. The actual size of the piece of memory should be returned in `*actual_sizep`. Should return `NULL` if it cannot allocate any more memory, or if it cannot reclaim the old block in a meaningful way. In that case, Prolog will use the other functions.

`free_hook`
: is called with a pointer to the piece of memory to be freed and its size. Should return non-zero iff the function was able to free this piece of memory. Otherwise, Prolog will keep using the memory as if it were not freed.

The default bottom layers look at the environment variables `PROLOGINITSIZE`, `PROLOGINCSIZE`, `PROLOGKEEPSIZE` and `PROLOGMAXSIZE`. They are useful to customize the default memory manager. If users redefine the bottom layer, they can choose to ignore these environment variables. See Section 1.1.2 [Environment Variables], page 10.

8.8.2 Loading Prolog Code

You can load your Prolog code compiled to Prolog object files into the system with the call `SP_load()`. This is the C equivalent of the Prolog predicate `load/1`:

```
int SP_load(char *filename)
```

Alternatively, you can restore a saved state with the call `SP_restore()`, which is the C equivalent of the Prolog predicate `restore/1`:

```
int SP_restore(char *filename)
```

`SP_load()` and `SP_restore()` return `SP_SUCCESS` for success or `SP_ERROR` if an error condition occurred.

Prolog error handling is mostly done by raising and catching exceptions. However, some *faults* are of a nature such that when they occur, the internal program state may be corrupted, and it is not safe to merely raise an exception. Memory allocation failures are examples of faults. In Runtime Systems, the following C macro provides an environment for handling faults:

```
int SP_on_fault(Stmt, Message, Cleanup)
```

which should occur in the scope of a `char *`*Message* declaration. *Stmt* is run, and if a fault occurs, *Stmt* is aborted, *Message* gets assigned a value explaining the fault, the Prolog internal

state is cleaned, all queries and SP_term_refs become invalid, and *Cleanup* run. If *Stmt* terminates normally, *Message* is left unchanged. For example, a "fault-proof" Runtime System could have the structure:

```
int main(int argc, char **argv)
{
  char *message;

  SP_initialize(argc, argv, "/usr/local/lib/sicstus37/bin");
loop:
  SP_on_fault(main_loop(), message,
              {printf("ERROR: %s\n",message); goto loop;});
  exit(0);
}

main_loop()
{...}
```

Faults that occur outside the scope of `SP_on_fault()` cause the Runtime System to halt with an error message.

The following function can be used to raise a fault. For example, it can be used in a signal handler for SIGSEGV to prevent the program from dumping core in the event of a segmentation violation (Runtime Systems have no predefined signal handling):

```
void SP_raise_fault(char *message)
```

8.8.3 Creating the Executable

This describes how to create a Runtime System with a set of statically linked foreign resources. For the SICStus library modules containing C-code, such linked foreign resources already exist in '`$SP_PATH/library`'. For user foreign code, you must first create linked foreign resources for them.

A utility command is provided for the generation of such systems. This command, located in '`$SP_PATH/bin`', is configurable as described in the Release Notes.

`spmkrs [--help] [-S]` *Executable* `[+c` *SourceFiles*`] [+o` *ObjectFiles*`] [+l` *Libraries*`] [+r` *Resources*`]`

> Generates a Runtime System with any foreign resources mentioned in *Resources* statically linked. The resources must be created with the `+static` option. Note however that a statically linked dynamic resource is not, in general, statically linked from the operating system's point of view. It must therefore be accessible at runtime as well as at build-time.
>
> Compiles *SourceFiles* and links the resulting object files with *ObjectFiles*, *Libraries*, and the foreign resources to produce *Executable*. The main program of the Runtime System must be provided by *SourceFiles* or *ObjectFiles*.

The option `-S` causes the SICStus Runtime Kernel to be statically linked into the executable, thus eliminating the need to load these at runtime. This produces a significantly larger executable. Observe that this option has no effect on the resources and libraries spefified after by `+r` and `+l`. **Note:** This option is not available on Win32.

The option `--help` prints out usage info.

For example, to compile and link a Runtime System '`ex_rt`' with the linked foreign resource `ex` statically linked, first create the linked foreign resource (see Section 8.2.7 [Creating the Linked Foreign Resource], page 135) using the `+static` option. Assuming the Runtime System's main program is in the file '`main.c`', then type:

```
% spmkrs ex_rt +c main.c +r ex
```

If you prefer not to use the utility commands, an alternative method is described in Chapter 38 [Runtime Utilities], page 373.

8.8.4 Running the Executable

To execute a Runtime System on a target machine you need:

the executable
: This is your application program created as described above.

the runtime kernel
: This is a shared object or a DLL, usually '`$SP_PATH/../libsprt37.so`' under UNIX, or '`%SP_PATH%\..\sprt37.dll`' under Windows.

the runtime library
: The saved state '`$SP_PATH/bin/sprt.sav`' contains the built-in predicates written in Prolog. It is restored into the program at runtime by the function `SP_initialize()`.

your Prolog code
: Either as a saved state, object code ('`.ql`' files), or source code ('`.pl`' files). They must be explicitly loaded by the program at runtime.

your linked foreign resources
: Any linked foreign resources which are not statically linked to the executable, including any linked foreign resources for library modules which are located in '`$SP_PATH/library`'.

Also, on most UNIX platforms, it is necessary to set an environment variable, typically `LD_LIBRARY_PATH`, for shared objects to be found at runtime. See Section 1.1.1 [Finding Libraries at Runtime], page 9.

8.9 Development Systems

This describes how to create a Development System with a set of linked foreign resources. For the SICStus library modules containing C-code, such linked foreign resources already exist in '`$SP_PATH/library`'. For user foreign code, you must first create linked foreign resources for them.

A utility command is provided for the generation of such systems. This command, located in '`$SP_PATH/bin`', is configurable as described in the Release Notes.

`spmkds [--help] [-S]` *Executable* `[`*Resources*`]`

Generates a Development System with any foreign resources mentioned in *Resources* statically linked. The resources must be created with the `+static` option. Note however that a statically linked dynamic resource is not, in general, statically linked from the operating system's point of view. It must therefore be accessible at runtime as well as at build-time.

The option `-S` causes the SICStus Runtime Kernel and development system extensions to be statically linked into the executable, thus eliminating the need to load these at runtime. This produces a significantly larger executable. Observe that this option has no effect on the resources. **Note:** This option is not available on Win32.

The option `--help` prints out usage info.

For example, to compile and link a Development System '`ex_stat`' with the linked foreign resource `ex` statically linked, first create the linked foreign resource (see Section 8.2.7 [Creating the Linked Foreign Resource], page 135) using the `+static` option. Then type:

```
% spmkds ex_stat ex
```

The executable can then be run as:

```
% sicstus -base ./ex_stat
```

and optionally more command-line options as for the standard `sicstus` command.

The generated executable will run the usual Prolog top level, and the full develoment environment is available as usual.

8.10 Examples

8.10.1 Train Example (connections)

This is an example of how to create a Runtime System. The Prolog program '`train.pl`' will display a route from one train station to another. The C program '`train.c`' calls the Prolog code and writes out all the routes found between two stations:

```
% train.pl

connected(From, To, Path) :-
        connected(From, To, Path, [From]).

connected(To, To, [To], _).
connected(From, To, [From|Path], Visited) :-
        (   connection(From, Via)
        ;   connection(Via, From)
        ),
        not_visited(Visited, Via),
        connected(Via, To, Path, [Via|Visited]).

connection('Stockholm', 'Katrineholm').
connection('Stockholm', 'Vasteras').
connection('Stockholm', 'Uppsala').
connection('Uppsala', 'Vasteras').
connection('Katrineholm', 'Hallsberg').
connection('Katrineholm', 'Linkoping').
connection('Hallsberg', 'Kumla').
connection('Hallsberg', 'Goteborg').
connection('Orebro', 'Vasteras').
connection('Orebro', 'Kumla').

not_visited([], _).
not_visited([X|Visited], Y) :- X \== Y, not_visited(Visited, Y).
```

```
/* train.c */

#include <stdio.h>
#include <sicstus/sicstus.h>

void write_path(SP_term_ref path)
{
  char *text = NULL;
  SP_term_ref
    tail = SP_new_term_ref(),
    via = SP_new_term_ref();

  SP_put_term(tail,path);

  while (SP_get_list(tail,via,tail))
  {
```

```
    if (text)
      printf(" -> ");

    SP_get_string(via, &text);
    printf("%s",text);
  }
  printf("\n");
}

int main(int argc, char **argv)
{
  int rval;
  SP_pred_ref pred;
  SP_qid goal;
  SP_term_ref from, to, path;

  /* Initialize Prolog engine. This call looks up SP_PATH in order to
   * find the Runtime Library.  */
  if (SP_FAILURE == SP_initialize(argc, argv, NULL))
    {
      fprintf(stderr, "SP_initialize failed: %s\n", SP_error_message(SP_errno));
      exit(1);
    }

  rval = SP_restore("train.sav");

  if (rval == SP_ERROR || rval == SP_FAILURE)
    {
      fprintf(stderr, "Could not restore \"train.sav\".\n");
      exit(1);
    }

  /* Look up connected/3. */
  if (!(pred = SP_predicate("connected",3,"")))
    {
      fprintf(stderr, "Could not find connected/3.\n");
      exit(1);
    }

  /* Create the three arguments to connected/3. */
  SP_put_string(from = SP_new_term_ref(), "Stockholm");
  SP_put_string(to = SP_new_term_ref(), "Orebro");
  SP_put_variable(path = SP_new_term_ref());

  /* Open the query. In a development system, the query would look like:
   *
   * | ?- connected('Stockholm','Orebro',X).
   */
```

```
  if (!(goal = SP_open_query(pred,from,to,path)))
    {
      fprintf(stderr, "Failed to open query.\n");
      exit(1);
    }

  /*
   * Loop through all the solutions.
   */
  while (SP_next_solution(goal))
    {
      printf("Path: ");
      write_path(path);
    }

  SP_close_query(goal);

  exit(0);
}
```

Create the saved-state containing the Prolog code:

```
% sicstus
SICStus 3.7: Tue Jun 02 12:29:02 MET DST 1998
| ?- compile(train).
{compiling /home/jojo/sicstus/examples/train.pl...}
{/home/jojo/sicstus/examples/train.pl compiled, 40 msec 2848 bytes}

yes
| ?- save_program('train.sav').
{SICStus state saved in /home/jojo/sicstus/examples/train.sav}

yes
| ?- halt.
```

Create the executable using `spmkrs`:

```
% spmkrs train +c train.c
SICStus 3.7: Tue Jun 02 12:29:02 MET DST 1998
{/var/tmp/aaa0ch9LZ.c generated, 0 msec}

yes
```

And finally, run the executable:

```
% ./train
Path: Stockholm -> Katrineholm -> Hallsberg -> Kumla -> Orebro
Path: Stockholm -> Vasteras -> Orebro
Path: Stockholm -> Uppsala -> Vasteras -> Orebro
```

8.10.2 I/O on Lists of Character Codes

This example is taken from the SICStus Prolog library (simplified, but operational). A stream for writing is opened where the written characters are placed in a buffer. When the stream is closed a list of character codes is made from the contents of the buffer. The example illustrates the use of user definable streams.

The `open_buf_stream()` function opens a stream where the characters are put in a buffer. The stream is closed by `stream_to_chars()` which returns the list constructed on the heap.

The Prolog code (simplified):

```
foreign(open_buf_stream, '$open_buf_stream'(-address('SP_stream'))).
foreign(stream_to_chars, '$stream_to_chars'(+address('SP_stream'),
                                            -term)).

foreign_resource(example, [open_buf_stream,stream_to_chars]).

:- load_foreign_resource(example).

%% with_output_to_chars(+Goal, -Chars)
%% runs Goal with current_output set to a list of characters

with_output_to_chars(Goal, Chars) :-
        '$open_buf_stream'(StreamCode),
        stream_code(Stream, StreamCode),
        current_output(CurrOut),
        set_output(Stream),
        call_and_reset(Goal, Stream, CurrOut, StreamCode, Chars).

call_and_reset(Goal, Stream, CurrOut, StreamCode, Chars) :-
        call(Goal), !,
        put(0),
        '$stream_to_chars'(StreamCode, Chars),
        reset_stream(Stream, CurrOut).
call_and_reset(_, Stream, CurrOut, _, _) :-
        reset_stream(Stream, CurrOut).

reset_stream(Stream, CurrOut) :-
        set_output(CurrOut),
        close(Stream).
```

The C code:

```
#include <sicstus/sicstus.h>

struct open_chars {
  char *chars;       /* character buffer */
  int index;         /* current insertion point */
```

```
  int size;
};

#define INIT_BUFSIZE 512

static int lputc(c, buf)
     int c;
     struct open_chars *buf;
{
  if (buf->index == buf->size)  /* grow buffer if necessary */
    {
      buf->size *= 2;
      buf->chars = (char *)realloc(buf->chars, buf->size);
    }
  return (buf->chars[buf->index++] = c);
}

static int lwclose(buf)
     struct open_chars *buf;
{
  free(buf->chars);
  free(buf);
  return 0;
}

void open_buf_stream(streamp)
     SP_stream **streamp;
{
  struct open_chars *buf;

  /* Allocate buffer, create stream & return stream code */

  buf = (struct open_chars *)malloc(sizeof(struct open_chars));
  SP_make_stream(buf, NULL, lputc, NULL, NULL, NULL, lwclose,
                 streamp);

  buf->chars = (char *)malloc(INIT_BUFSIZE);
  buf->size = INIT_BUFSIZE;
  buf->index = 0;
}

void stream_to_chars(streamp, head)
     SP_stream *streamp;
     SP_term_ref head;
{
  SP_term_ref tail = SP_new_term_ref();
  struct open_chars *buf = (struct open_chars *)streamp->user_handle;
```

```
  /* Make a list of character codes out of the buffer */

  SP_put_string(tail, "[]");
  SP_put_list_chars(head, tail, buf->chars);
}
```

8.10.3 Exceptions from C

Consider, for example, a function which returns the square root of its argument after checking that the argument is valid. If the argument is invalid, the function should raise an exception instead.

```
/* math.c */

#include <math.h>
#include <stdio.h>
#include <sicstus/sicstus.h>

double sqrt_check(d)
     double d;
{
  if (d < 0.0)
    {    /* build a domain_error/4 exception term */
      SP_term_ref culprit=SP_new_term_ref();
      SP_term_ref argno=SP_new_term_ref();
      SP_term_ref expdomain=SP_new_term_ref();
      SP_term_ref t1=SP_new_term_ref();

      SP_put_float(culprit, d);
      SP_put_integer(argno, 1);
      SP_put_string(expdomain, ">=0.0");
      SP_cons_functor(t1, SP_atom_from_string("sqrt"), 1, culprit);
      SP_cons_functor(t1, SP_atom_from_string("domain_error"), 4,
                      t1, argno, expdomain, culprit);
      SP_raise_exception(t1);    /* raise the exception */
      return 0.0;
    }
  return sqrt(d);
}
```

The Prolog interface to this function is defined in a file '`math.pl`'. The function uses the `sqrt()` library function, and so the math library '`-lm`' has to be included:

```
/* math.pl */

foreign_resource(math, [sqrt]).

foreign(sqrt_check, c, sqrt(+float, [-float])).
```

```
:- load_foreign_resource(math).
```

A linked foreign resource is created:

```
% splfr math math.pl +c math.c +l -lm
```

A simple session using this function could be:

```
% sicstus
SICStus 3  #5: Sat Feb 24 00:35:37 MET 1996
| ?- [math].
{consulting /home/san/pl/math.pl...}
{/home/san/pl/math.pl consulted, 10 msec 816 bytes}

yes
| ?- sqrt(5.0,X).

X = 2.23606797749979 ?

yes
| ?- sqrt(a,X).
{TYPE ERROR: sqrt(a,_30) - arg 1: expected number, found a}
| ?- sqrt(-5,X).
{DOMAIN ERROR: sqrt(-5.0) - arg 1: expected '>=0.0', found -5.0}
```

The above example used the foreign language interface with dynamic linking. To statically link '`math.o`' with the Prolog emulator, the following steps would have been taken:

```
% splfr math math.pl +static +c math.c +l -lm
SICStus 3  #5: Sat Feb 24 00:35:37 MET 1996
{/tmp/00249aaa.c generated, 10 msec}

yes
% spmkds mathsp math
SICStus 3  #5: Sat Feb 24 00:35:37 MET 1996
{/tmp/00274aaa.c generated, 0 msec}

yes
% sicstus -base ./mathsp
SICStus 3  #5: Sat Feb 24 00:35:37 MET 1996
| ?- [math].
{consulting /home/san/pl/math.pl...}
{/home/san/pl/math.pl consulted, 0 msec 864 bytes}

yes
| ?- sqrt(5.0,X).

X = 2.23606797749979 ?
```

```
yes
```

8.10.4 Stream Example

This is a small example how to initialize a bidirectional socket stream (error handling omitted):

```
typedef struct {
  int fd;                       /* socket number */
  FILE *r_stream;               /* For reading */
  FILE *w_stream;               /* For writing */
} SocketData;

int socket_sgetc(SocketData *socket)
{
  return fgetc(socket->r_stream);
}

int socket_sputc(char c, SocketData *socket)
{
  return fputc(c, socket->w_stream);
}

int socket_sflush(SocketData *socket)
{
  return fflush(socket->w_stream);
}

int socket_seof(SocketData *socket)
{
  return feof(socket->r_stream);
}

void socket_sclrerr(SocketData *socket)
{
  clearerr(socket->r_stream);
  clearerr(socket->w_stream);
}

int socket_sclose(SocketData *socket)
{
  fclose(socket->r_stream);
  fclose(socket->w_stream);
  close(socket->fd);
  free(socket);
  return 0;
}
```

```
SP_stream *new_socket_stream(int fd)
{
  SP_stream *stream;
  SocketData *socket;

/* Allocate and initialize data local to socket */

  socket = (SocketData *)malloc(sizeof(SocketData));
  socket->fd = fd;
  socket->r_stream = fdopen(fd,"r");
  socket->w_stream = fdopen(fd,"w");

/* Allocate and initialize Prolog stream */

  SP_make_stream(
                socket,
                socket_sgetc,
                socket_sputc,
                socket_sflush,
                socket_seof,
                socket_sclrerr,
                socket_sclose,
                &stream);

/* Allocate and copy string */

  stream->filename = "socket";
  stream->fd = fd;

  return stream;
}
```

9 Mixing Java and Prolog

Jasper is a bi-directional interface between programs written in Java and programs written in Prolog. The Java-side of the interface constists of a Java package (`jasper`) containing classes representing the SICStus emulator. The Prolog part is designed as an extension to the foreign language interface, which means that `foreign/3` declarations are used to map Java-methods on Prolog predicates and provide automatic conversion of arguments.

Jasper can be operated in two different ways, depending on which emulator is used as top-level application; the Java Runtime System or the SICStus Development System. When the Java Runtime System is used as a top-level application, the SICStus runtime kernel is loaded into the Java Runtime System. When the Java Runtime System is loaded into the SICStus Development System, it is loaded as any other foreign resource, i.e. by using `load_foreign_resource/1`. This distinction is not visible to the programmer, but it is useful to have a picture of how the interface works.

Note: Some of the information in this chapter is a recapitulation of the information in the chapter Chapter 8 [Mixing C and Prolog], page 127. The intention is that this chapter should be possible to read fairly independently.

9.1 Prerequisites

The low-level interface uses the JNI (Java Native Interface) to call C functions from Java. The JNI is a native interface standard developed by Sun Microsystems. See Section 9.7 [Java Resources Online], page 176. Since the interface uses the JNI, it is necessary that the Java VM which is to be used has support for JNI. Even though the JNI is intended as a standard, not all vendors support it. If your Java installation does not support the JNI, Sun's *Java Runtime Environment* provides a minimal execution enviroment for running Java code. It can be downloaded from

```
http://java.sun.com/products/jdk/1.1/jre/index.html
```

The rest of this chapter assumes that there is a Java installation with JNI support available.

9.2 Calling Java from Prolog

Java methods are called from Prolog much in the same way as C functions are called (see Section 8.2 [Calling C], page 128); by creating a *foreign resource*. When loaded, this resource installs a set of predicates which are mapped onto Java-methods and when invoked, converts the Prolog arguments to the corresponding Java-types before calling the Java method itself.

In fact, a foreign resource (as defined in Section 8.2.1 [Foreign Resource], page 128) is not language specific itself. The language is instead specified in the second argument to the `foreign/3` fact and it is possible to mix foreign C functions with foreign Java methods.

How a foreign resource is created in general is described in detail in Section 8.2.7 [Creating the Linked Foreign Resource], page 135. The following section(s) will focus on the Java-specific parts of foreign resources.

9.2.1 Static and Dynamic Linking

There is no support for static linking of foreign resources containing Java declarations, since Java implementations usually do not support static linking.

9.2.2 Declarating Java-methods

Java-methods are declared similarly to C-functions. There are two major differences. The first is how methods are identified. It is not enough to simply use an atom as the C interface does. Instead, a term `method/3` is introduced:

`method(+`*ClassName*`,+`*MethodName*`,+`*Flags*`)`
: Used as first argument to `foreign/3` when declaring Java methods. The first argument is an atom containing the *Fully Qualified Classname* of the class. The second argument is the method name. The third argument is a list of flags. Possible flags are `instance` or `static`, indicating whether or not the method is static or non-static. Non-static methods *must* have an object-reference as their first argument. This is a reference to the object on which the method will be invoked.

  This term is then used to identify the method in the `foreign_resource/2` predicate. So, to define a foreign resource exporting the non-static Java method `getFactors` in the class `PrimeNumber` in the package `numbers`, the `method/3` term would look like

```
method('numbers/PrimeNumber','getFactors',[instance])
```

The syntax for `foreign/3` is the basically the same as for C-functions:

`foreign(+`*MethodIdentifier*`, java, +`*Predicate*`)`
: A *hook predicate*, specifies the Prolog interface to a Java method. *MethodIdentifier* is `method/3` term as described above. *Predicate* specifies the name of the Prolog predicate that will be used to call *MethodIdentifier*. *Predicate* also specifies how the predicate arguments are to be translated into the corresponding Java arguments.

9.3 Conversions between Prolog Arguments and Java Types

The following table lists the possible values of arguments of the predicate specification to `foreign/3`. The value declares which conversion between corresponding Prolog argument and Java type will take place.

Note: The conversion declarations (composed of the declarators specified below) together with the `method/3` term are used by the glue-code generator to create the method's *type signature*, i.e. a string which can uniquely identify a method within a class. This means that unlike the C interface, the conversion declarations for a Java method will affect the lookup of the method-name (in the C interface, only the function *name* is relevant). So, if a method is declared as `foo(+integer)`, there must be a method which has the name `foo` and takes one argument of type `int`, or an argument which can be automatically converted to an `int` (a `short`, for example).

`Prolog: +integer`

`Java: int`

The argument should be a number. It is converted to a Java `int`.

`Prolog: +byte`

`Java: byte`

The argument should be a number. It is converted to a Java `byte`.

`Prolog: +short`

`Java: short`

The argument should be a number. It is converted to a Java `short`.

`Prolog: +long`

`Java: long`

The argument should be a number. It is converted to a Java `long`.

Note: Since Java's `long` type is 64 bits wide and there is no standardized support for 64 bits integers in C, the value will be truncated. So, this declaration is really only useful in order to indicate which method should be used. For example:

```
class Bar
{
  void foo(int x)
    { ... }

  void foo(long x)
    { ... }
}
```

In order to be able to indicate that the latter of the `foo` methods should be called, a `+long` declaration must be used, even if the value itself will be truncated in the call.

`Prolog: +float`

`Java: float`

The argument should be a number. It is converted to a Java `float`.

`Prolog: +double`

`Java: double`

The argument should be a number. It is converted to a Java `double`.

`Prolog: +term`

`Java: SPTerm`

The argument can be any term. It is passed to Java as an object of the class `SPTerm`.

`Prolog: +object`

`Java: SPTerm`

The argument should be a the Prolog representation of a Java object. Unless it is the first argument in a non-static method (in which case is it treated as the object on which the method should be invoked), it is passed to the Java method as an object of class `SPTerm`.

`Prolog: +atom`

`Java: SPTerm`
: The argument should be an atom. The Java method will be passed an object of class `SPTerm`.

`Prolog: +boolean`

`Java: boolean`
: The argument should be an atom in `{true,false}`. The Java method will receive a `boolean`.

`Prolog: +chars`

`Java: String`
: The argument should be a list of character codes. The Java method will receive an object of class `String`.

`Prolog: +string`

`Java: String`
: The argument should be an atom. The Java method will receive an object of class `String`.

`Prolog: -atom`

`Java: SPTerm`
: The argument should be an unbound variable. The Java method will receive an atom of class `SPTerm` which can be modified. The argument will be bound to the value of the atom when the method returns.

`Prolog: -chars`

`Java: StringBuffer`
: The argument should be an unbound variable. The Java method will receive an object of type `StringBuffer` which can be modified. The argument will be bound to a list of the character codes of the `StringBuffer` object.

`Prolog: -string`

`Java: StringBuffer`
: The argument should be an unbound variable. The Java method will receive an object of type `StringBuffer` which can be modified. The argument will be bound to an atom converted from the `StringBuffer` object.

`Prolog: [-integer]`

`Java: int` *M*`()`
: The Java method should return an `int`. The value will be converted to a Prolog integer.

`Prolog: [-byte]`

`Java: byte` *M*`()`
: The Java method should return a `byte`. The value will be converted to a Prolog integer.

`Prolog: [-short]`

`Java: short` *M*`()`
: The Java method should return a `short`. The value will be converted to a Prolog integer.

`Prolog: [-long]`

`Java: long` *M*`()`
: The Java method should return a `long`. The value will be converted and possibly truncated to a Prolog integer.

`Prolog: [-float]`

`Java: float` *M*`()`
: The Java method should return a `float`. The value will be converted to a Prolog float.

`Prolog: [-double]`

`Java: double` *M*`()`
: The Java method should return a `double`. The value will be converted to a Prolog float.

`Prolog: [-term]`

`Java: SPTerm` *M*`()`
: The Java method should return an object of class `SPTerm` which will be converted to a Prolog term.

`Prolog: [-object]`

`Java: SPTerm` *M*`()`
: The Java method should return an object of class `SPTerm` which will be converted to the internal Prolog representation of a Java object.

`Prolog: [-atom]`

`Java: SPTerm` *M*`()`
: The Java method should return an object of class `SPTerm` which will be converted to a Prolog atom.

`Prolog: [-boolean]`

`Java: boolean` *M*`()`
: The Java should return a `boolean`. The value will be converted to a Prolog atom in `{true,false}`.

`Prolog: [-chars]`

`Java: String` *M*`()`
: The Java method should return an object of class `String` which will be converted to a list of character codes.

`Prolog: [-string]`

`Java: String` *M*`()`
: The Java method should return an object of class `String` which will be converted to an atom.

9.3.1 Calling Java from Prolog: An Example

The following is an simple, but complete example of how a Java method can be called from Prolog.

First, we must write the resource file. Let us call it '`simple.pl`'.

```
% File: simple.pl

:- module(simple, [simple/2]).

:- use_module(library(jasper)).
:- load_foreign_resource(simple).

foreign(method('Simple', 'simpleMethod', [static]), java,
        simple(+integer,[-integer])).

foreign_resource(simple,
                 [
                  method('Simple', 'simpleMethod', [static])
                 ]).
```

This file is the processed with the script `splfr` (see Section 8.2.5 [Interface Predicates], page 133) to produce a foreign resource:

```
% splfr simple simple.pl
SICStus  3.7: Mon Mar  2 19:22:44 MET 1998
{/var/tmp/aaaa004Cd.c generated, 20 msec}

yes
```

Note that we do not specify any Java files to `splfr` as we would specify C files when building foreign resources for C code. This is because the C code can be compiled into the resource itself, while the Java code must be loaded at runtime into the JVM. This means that the resource will only contain the glue-code for calling the JVM, and no actual Java code. Hence, these resources are usually quite small.

Note also that the use of `+dynamic` is implicit and obligatory; resources with Java-calls *must* be dynamic.

Now, we need some Java code to call:

```
Simple.java:

public class Simple
{
  static int simpleMethod(int value)
  {
    return value*42;
  }
}
```

This Java code must now be compiled. Refer to the documentation of your Java implementation exactly how to do this. On Solaris, this might look like:

```
% javac Simple.java
```

Now we are ready to call the method `simple/2` from inside SICStus.

```
% sicstus
SICStus  3.7: Mon Mar  2 19:22:44 MET 1998
| ?- compile(simple).
{compiling ...}
[...]
{compiled ... simple.pl in module simple, 160 msec 48640 bytes}

yes
| ?- simple(17,X).

X = 714 ?

yes
| ?-
```

What has happened is that the predicate `simple/2` has been installed as a predicate defined in Java (this is not exactly true; the predicate is defined as a C-function which calls the Java method). When we load the `simple` module, we will first load the `jasper` module (and thereby the JVM) and then load the `simple` foreign resource, which defines the `simple/2` predicate.

9.4 Calling Prolog from Java

Calling Prolog from Java is done by using the Java package `jasper`. This package contains a set of Java classes which can be used to create and manipulate terms, ask queries and request one or more solutions. The functionality provided by this set of classes is basically the same as the functionality provided by the C-Prolog interface (see Chapter 8 [Mixing C and Prolog], page 127).

The usage is easiest described by an example. The following is a Java version of the `train` example. See Section 8.10.1 [Train], page 157.

```
import jasper.*;

public class Simple
{
  public static void main(String argv[]) {

  SICStus sp;
  SPPredicate pred;
  SPTerm from, to, way;
  SPQuery query;
  int i;
```

```
    try
      {
        sp = new SICStus(argv,null);

        sp.load("train.ql");

        pred = new SPPredicate(sp, "connected", 4, "");
        to = new SPTerm(sp, "Orebro");
        from = new SPTerm(sp, "Stockholm");
        way = new SPTerm(sp).putVariable();

        query = sp.openQuery(pred, new SPTerm[] { from, to, way, way });

        while (query.nextSolution())
          {
            System.out.println(way.toString());
          }
      }
    catch ( Exception e )
      {
        e.printStackTrace();
      }
    }
  }
```

This is how it works:

1. First, we import all the `jasper`-classes. To find them, the class-path should be set to contain the path to the SICStus libraries (`$SP_PATH/library`). How this is done depends on the operating system and which Java implementation is used, but typically there is a environment variable `CLASSPATH` which contains the path. The path can often also be set on the commandline using an option such as '`-classpath`'. Refer to your Java documentation for more details.
2. Before any predicates can be called, the SICStus emulator must be initialized. This is done by instantiating the `SICStus` class. NOTE: This class must only be instantiated once per Java process. Multiple SICStus-objects are not supported.

   Most methods take a reference to this object as their first argument. This is implicit in the rest of this chapter, unless otherwise stated.
3. The next step is to load the Prolog code. This is done by the method `load`. Corresponds to `SP_load()` in the C-interface. See Section 8.8.2 [Loading Prolog Code], page 153.
4. Now, everything is set up to start making queries. In order to make a query, the actual query term must be created. This is done by creating an object of the `SPPredicate` class:

   ```
   SPPredicate pred = new SPPredicate(sp, "connected", 4, "");
   ```

5. At this point, we have created a predicate object for the predicate `connected/4`. It is now time to create the arguments for the query. The arguments are placed in an array which is passed to a suitable method to make the query.

   The arguments consist of objects of the class `SPTerm`. For example, if we need two atoms and a variable for the query

   ```
   | ?- connected('Stockholm', 'Orebro', X, X).
   ```

   the following Java code will do it for us:

   ```
   to = new SPTerm(sp, "Orebro");
   from = new SPTerm(sp, "Stockholm");
   way = new SPTerm(sp).putVariable();
   ```

6. Now it is time to make the query. As in the C-Prolog interface, there are three ways of making a query.

   `query(pred, args)`
   : This method is useful if you are only interested in finding the first solution to a goal. In the case of `connected/4` this is not the case; there are more than one solution.

   `queryCutFail(pred, args)`
   : This method is useful if you are only interested in the side-effects of the query. As `query()` it only finds the first solution, and then it cuts away all other solutions and fails.

   `openQuery(pred, args)`
   : This method is useful when you are interested in some or all solutions to the query. Since the `connected/4` may give us multiple solution, this is what we will use.

     ```
     SPQuery query;

     query = sp.openQuery(pred, new SPTerm[] { from, to, way, way });

     while (query.nextSolution())
       System.out.println(way.toString());
     ```

     The `openQuery` method returns a reference to the query, an object of the `SPQuery` class. To obtain solutions, the method `nextSolution()` is called with no arguments. `nextSolution` returns `true` as long as there are more solutions, the example above will print the value of `way` until there are no more solutions.

9.5 Jasper Package Class Reference

Detailed documentation of the classes in the `jasper` package can be found at

```
http://www.sics.se/isl/sicstus/jasper/Package-jasper.html
```

This documentation will most probably be included in a future version of the manual.

9.6 Exception Handling

Exceptions are handled seamlessly between Java and Prolog. This means that exceptions can be thrown in Prolog and caught in Java and the other way around. For example, if a predicate called from Java raises an exception by `raise_exception/1` and the predicate itself does not catch the exception, the Java-method which performed the query, `queryCutFail()` for example, will throw an exception containing the exception term. Symmetrically, a Java-exception thrown (and not caught) in a method called from Prolog will cause the corresponding predicate (`simple/2` in the example above) to raise an exception containing the exception object (in the internal Prolog representation of a Java object).

9.7 Resources

There are almost infinitely many Java resources on the Internet. Here is a list of a few which are related to Jasper:

- The SICStus Prolog Release Notes contains information useful in order to get Jasper working.
- There is a collection of Java-to-Prolog examples in

  ```
  library('jasper/examples')
  ```

  See the file '`README`' for details on how to use the examples.
- The main Java resource is Sun's own Java homepage:

  ```
  http://java.sun.com
  ```
- Yahoo:

  ```
  http://www.yahoo.com/Computers_and_Internet/Programming_Languages/Java/
  ```
- Information about the JNI can be found at:

  ```
  http://java.sun.com/products/jdk/1.1/docs/guide/jni/index.html
  ```
- The ACM student magazine *Crossroads* has published an article on the JNI, available online at:

  ```
  http://www1.acm.org:82/crossroads/xrds4-2/jni.html
  ```

10 Programming Tips and Examples

This chapter describes how to write clean programs that will execute efficiently. To some extent, writing efficient code in any language requires basic knowledge of its compiler, and we will mention some important properties of the SICStus Prolog compiler. A number of simple examples of Prolog programming are also given.

10.1 Programming Guidelines

A lot of clarity and efficiency is gained by sticking to a few basic rules. This list is necessarily very incomplete. The reader is referred to textbooks such as [O'Keefe 90] for a thorough exposition of the elements of Prolog programming style and techniques.

- Don't write code in the first place if there is a library predicate that will do the job.
- Write clauses representing base case before clauses representing recursive cases.
- Input arguments before output arguments in clause heads and goals.
- Use pure data structures instead of data base changes.
- Use cuts sparingly, and *only* at proper places (see Section 2.5 [Cut], page 36). A cut should be placed at the exact point that it is known that the current choice is the correct one: no sooner, no later.
- Make cuts as local in their effect as possible. If a predicate is intended to be deterministic, define *it* as such; do not rely on its callers to prevent unintended backtracking.
- Binding output arguments before a cut is a common source of programming errors, so don't do it.
- Replace cuts by if-then-else constructs if the test is simple enough (see Section 10.4 [If-Then-Else], page 178).
- Use disjunctions sparingly, *always* put parentheses around them, *never* put parentheses around the individual disjuncts, *never* put the '`;`' at the end of a line.
- Write the clauses of a predicate so that they discriminate on the principal functor of the first argument (see below). For maximum efficiency, avoid "defaulty" programming ("catch-all" clauses).
- Don't use lists (`[...]`), "round lists" (`(...)`), or braces (`{...}`) to represent compound terms, or "tuples", of some fixed arity. The name of a compound term comes for free.

10.2 Indexing

The clauses of any predicate are *indexed* according to the principal functor of the first argument in the head of the clause. This means that the subset of clauses which match a given goal, as far as the first step of unification is concerned, is found very quickly, in practically constant time. This can be very important where there is a large number of clauses for a predicate. Indexing also improves the Prolog system's ability to detect determinacy—important for conserving working storage, and strongly related to last call optimization (see below).

Indexing applies to interpreted clauses as well as to compiled clauses.

10.3 Last Call Optimization

The compiler incorporates *last call optimization* to improve the speed and space efficiency of determinate predicates.

When execution reaches the last goal in a clause belonging to some predicate, and provided there are no remaining backtrack points in the execution so far of that predicate, all of the predicate's local working storage is reclaimed *before* the final call, and any structures it has created become eligible for garbage collection. This means that programs can now recurse to arbitrary depths without necessarily exceeding core limits. For example:

```
cycle(State) :- transform(State, State1), cycle(State1).
```

where `transform/2` is a determinate predicate, can continue executing indefinitely, provided each individual structure, *State*, is not too large. The predicate cycle is equivalent to an iterative loop in a conventional language.

To take advantage of last call optimization one must ensure that the Prolog system can recognize that the predicate is determinate at the point where the recursive call takes place. That is, the system must be able to detect that there are no other solutions to the current goal to be found by subsequent backtracking. In general this involves reliance on the Prolog compiler's indexing and/or use of cut; see Section 2.5 [Cut], page 36.

10.4 If-Then-Else Compilation

Ordinary disjunction, `(`*P*`;`*Q*`)`, is treated by the compiler as an anonymous predicate with two clauses, and the execution of a disjunction relies on backtracking to explore the two disjuncts.

If-then-else statements of the form:

`(`*If* `->` *Then*`;` *Else*`)`

are recognized by the compiler and are under certain conditions compiled to code that is much more efficient than the corresponding disjunction, essentially turning the *If* test to a conditional jump and often avoiding costly backtracking altogether.

For this optimization to be effective, the test must be a conjunction of a restricted set of built-in predicates (roughly, arithmetic tests, type tests and term comparisons).

This optimization is actually somewhat more general than what is described above. A sequence of *guarded clauses*:

Head1 `:-` *Guard1*`, !,` *Body1*`.`
`...`
Headm `:-` *Guardm*`, !,` *Bodym*`.`
Headn `:-` *Bodym*`.`

is eligible for the same optimization, provided that the arguments of the clause heads are all unique variables and that the "guards" are simple tests as described above.

10.5 Programming Examples

The rest of this chapter contains a number of simple examples of Prolog programming, illustrating some of the techniques described above.

10.5.1 Simple List Processing

The goal `concatenate(`*L1*`,`*L2*`,`*L3*`)` is true if list *L3* consists of the elements of list *L1* concatenated with the elements of list *L2*. The goal `member(`*X*`,`*L*`)` is true if *X* is one of the elements of list *L*. The goal `reverse(`*L1*`,`*L2*`)` is true if list *L2* consists of the elements of list *L1* in reverse order.

```
concatenate([], L, L).
concatenate([X|L1], L2, [X|L3]) :- concatenate(L1, L2, L3).

member(X, [X|_]).
member(X, [_|L]) :- member(X, L).

reverse(L, L1) :- reverse_concatenate(L, [], L1).

reverse_concatenate([], L, L).
reverse_concatenate([X|L1], L2, L3) :-
        reverse_concatenate(L1, [X|L2], L3).
```

10.5.2 Family Example (descendants)

The goal `descendant(`*X*`,`*Y*`)` is true if *Y* is a descendant of *X*.

```
descendant(X, Y) :- offspring(X, Y).
descendant(X, Z) :- offspring(X, Y), descendant(Y, Z).

offspring(abraham, ishmael).
offspring(abraham, isaac).
offspring(isaac, esau).
offspring(isaac, jacob).
```

If for example the query

```
| ?- descendant(abraham, X).
```

is executed, Prolog's backtracking results in different descendants of Abraham being returned as successive instances of the variable *X*, i.e.

```
X = ishmael
X = isaac
X = esau
X = jacob
```

10.5.3 Association List Primitives

These predicates implement "association list" primitives. They use a binary tree representation. Thus the time complexity for these predicates is *O(lg N)*, where *N* is the number of keys. These predicates also illustrate the use of `compare/3` (see Section 7.3 [Term Compare], page 97) for case analysis.

The goal `get_assoc(`*Key*`,` *Assoc*`,` *Value*`)` is true when *Key* is identical to one of the keys in *Assoc*, and *Value* unifies with the associated value.

```
get_assoc(Key, t(K,V,L,R), Val) :-
        compare(Rel, Key, K),
        get_assoc(Rel, Key, V, L, R, Val).

get_assoc(=, _, Val, _, _, Val).
get_assoc(<, Key, _, Tree, _, Val) :-
        get_assoc(Key, Tree, Val).
get_assoc(>, Key, _, _, Tree, Val) :-
        get_assoc(Key, Tree, Val).
```

The goal `put_assoc(`*Key*`,` *OldAssoc*`,` *Val*`,` *NewAssoc*`)` is true when *OldAssoc* and *NewAssoc* define the same mapping for all keys other than *Key*, and `get_assoc(`*Key*`,` *NewAssoc*`,` *Val*`)` is true.

```
put_assoc(Key, t, Val, Tree) :- !, Tree = t(Key,Val,t,t).
put_assoc(Key, t(K,V,L,R), Val, New) :-
        compare(Rel, Key, K),
        put_assoc(Rel, Key, K, V, L, R, Val, New).

put_assoc(=, Key, _, _, L, R, Val, t(Key,Val,L,R)).
put_assoc(<, Key, K, V, L, R, Val, t(K,V,Tree,R)) :-
        put_assoc(Key, L, Val, Tree).
put_assoc(>, Key, K, V, L, R, Val, t(K,V,L,Tree)) :-
        put_assoc(Key, R, Val, Tree).
```

10.5.4 Differentiation

The goal `d(`*E1*, *X*, *E2*`)` is true if expression *E2* is a possible form for the derivative of expression *E1* with respect to *X*.

```
:- op(300, xfy, **). /* binds tighter than * */

d(X, X, D) :- atomic(X), !, D = 1.
d(C, X, D) :- atomic(C), !, D = 0.
d(U+V, X, DU+DV) :- d(U, X, DU), d(V, X, DV).
d(U-V, X, DU-DV) :- d(U, X, DU), d(V, X, DV).
d(U*V, X, DU*V+U*DV) :- d(U, X, DU), d(V, X, DV).
d(U**N, X, N*U**N1*DU) :- integer(N), N1 is N-1, d(U, X, DU).
d(-U, X, -DU) :- d(U, X, DU).
```

10.5.5 Use of Meta-Logical Predicates

This example illustrates the use of the meta-logical predicates `var/1`, `arg/3`, and `functor/3` (see Section 7.7 [Meta Logic], page 110). The procedure call `variables(`*Term*, *L*, `[])` instantiates variable *L* to a list of all the variable occurrences in the term *Term*. e.g.

```
| ?- variables(d(U*V, X, DU*V+U*DV), L, []).

L = [U,V,X,DU,V,U,DV]
```

```
variables(X, [X|L0], L) :- var(X), !, L = L0.
variables(T, L0, L) :-
%       nonvar(T),
        functor(T, _, A),
        variables(0, A, T, L0, L).

variables(A, A, _, L0, L) :- !, L = L0.
variables(A0, A, T, L0, L) :-
%       A0<A,
        A1 is A0+1,
        arg(A1, T, X),
        variables(X, L0, L1),
        variables(A1, A, T, L1, L).
```

10.5.6 Use of Term Expansion

This example illustrates the use of `user:term_expansion/(2,4)` to augment the built-in predicate `expand_term/2` which works as a filter on the input to compile and consult. The code below will allow the declaration '`:- wait f/3`' as an alias for '`:- block f(-,?,?)`'. Wait declarations were used in previous versions of SICStus Prolog.

Note the `multifile` declaration, which prevents this `user:term_expansion/(2,4)` clause from erasing any other clauses for the same predicate that might have been loaded.

```
:- op(1150, fx, [wait]).

:- multifile user:term_expansion/2.
user:term_expansion((:- wait F/N), (:- block Head)) :-
        functor(Head, F, N),
        wb_args(N, Head).

wb_args(1, Head) :- !, arg(1, Head, -).
wb_args(N, Head) :-
%       N>1,
        arg(N, Head, ?),
        N1 is N-1,
        wb_args(N1, Head).
```

10.5.7 Prolog in Prolog

This example shows how simple it is to write a Prolog interpreter in Prolog, and illustrates the use of a variable goal. In this mini-interpreter, goals and clauses are represented as ordinary Prolog data structures (i.e. terms). Terms representing clauses are specified using the predicate `my_clause/1`, e.g.

```
my_clause( (grandparent(X, Z) :- parent(X, Y), parent(Y, Z)) ).
```

A unit clause will be represented by a term such as

```
my_clause( (parent(john, mary) :- true) ).
```

The mini-interpreter consists of three clauses:

```
execute((P,Q)) :- !, execute(P), execute(Q).
execute(P) :- predicate_property(P, built_in), !, P.
execute(P) :- my_clause((P :- Q)), execute(Q).
```

The second clause enables the mini-interpreter to cope with calls to ordinary Prolog predicates, e.g. built-in predicates. The mini-interpreter needs to be extended to cope with the other control structures, i.e. `!`, `(P;Q)`, `(P->Q)`, `(P->Q;R)`, `(\+ P)`, and `if(P,Q,R)`.

10.5.8 Translating English Sentences into Logic Formulae

The following example of a definite clause grammar defines in a formal way the traditional mapping of simple English sentences into formulae of classical logic. By way of illustration, if the sentence

> Every man that lives loves a woman.

is parsed as a sentence by the call

```
| ?- phrase(sentence(P), [every,man,that,lives,loves,a,woman]).
```

then P will get instantiated to

```
all(X):(man(X)&lives(X) => exists(Y):(woman(Y)&loves(X,Y)))
```

where `:`, `&` and `=>` are infix operators defined by

```
:- op(900, xfx, =>).
:- op(800, xfy, &).
:- op(300, xfx, :).
```

The grammar follows:

```
sentence(P) --> noun_phrase(X, P1, P), verb_phrase(X, P1).

noun_phrase(X, P1, P) -->
        determiner(X, P2, P1, P), noun(X, P3), rel_clause(X, P3, P2).
noun_phrase(X, P, P) --> name(X).

verb_phrase(X, P) --> trans_verb(X, Y, P1), noun_phrase(Y, P1, P).
verb_phrase(X, P) --> intrans_verb(X, P).

rel_clause(X, P1, P1&P2) --> [that], verb_phrase(X, P2).
rel_clause(_, P, P) --> [].

determiner(X, P1, P2, all(X):(P1=>P2)) --> [every].
determiner(X, P1, P2, exists(X):(P1&P2)) --> [a].

noun(X, man(X)) --> [man].
noun(X, woman(X)) --> [woman].

name(john) --> [john].

trans_verb(X, Y, loves(X,Y)) --> [loves].
intrans_verb(X, lives(X)) --> [lives].
```

10.5.9 Muse FLI Example

This example illustrates the techniques used in foreign language interface programming in Muse.

Let us assume we want to count the number of solutions of some problem, as well as the number of solutions produced by each worker. The following C file, '`count.c`', defines some procedures that can be used for this purpose.

```
#include <sicstus/sicstus.h>

static int *overall_counter_lock;
static int *worker_counters, *overall_counter;


void allocate_counters()
{
  overall_counter_lock = (int *)SP_malloc(sizeof(int));
  overall_counter = (int *)SP_malloc(sizeof(int));
  worker_counters = (int *)SP_malloc(muse_max_workers()*sizeof(int));
}
```

```
void init_counters()
{
  int i;

  muse_init_lock(overall_counter_lock);
  *overall_counter = 0;

  for (i=0; i<muse_num_workers(); i++)
    worker_counters[i]=0;
}

long get_counter(c)
     long c;
{
  if (c >= 0) return worker_counters[c];
  else        return *overall_counter;
}

long incr_counter()
{
  int i;

  muse_lock(overall_counter_lock);
  i = ++(*overall_counter);
  muse_un_lock(overall_counter_lock);
  ++(worker_counters[muse_worker_id()]);

  return i;
}
```

We have defined here an `overall_counter` and an array of `worker_counters`. There is a lock, `overall_counter_lock`, ensuring the atomicity of incrementing the `overall_counter`. The code `allocate_counter` allocates the counters in shared memory. The `init_counter` procedure initializes all counters, while `get_counters` retrieves the current value of a counter associated with a worker (non-negative argument), or the value of the `overall_counter` (negative argument). The `incr_counter` procedure increments the counter for the worker, as well as the overall counter.

The following Muse code, '`count.pl`', makes use of the above C routines to accomplish the task of counting the solutions of some goal.

```
foreign_resource(count, [allocate_counters,init_counters,
                         incr_counter,get_counter]).

foreign(allocate_counters, c, allocate_counters).
foreign(init_counters, c, init_counters).
foreign(incr_counter, c, incr_counter([-integer])).
foreign(get_counter, c, get_counter(+integer, [-integer])).
```

```
:- load_foreign_resource(count).
:- allocate_counters.

count(Goal) :-
        statistics(walltime, [Start,_]),
        count_solutions(Goal),
        statistics(walltime, [End,_]),
        display_counts(N),
        Time is End - Start,
        format('Total ~d solutions in ~3d seconds.~n', [N,Time]).

count_solutions(Goal) :-
        init_counters,
        Goal,
        incr_counter(_),
        fail.
count_solutions(_).

display_counts(N) :-
        get_counter(-1, N),
        muse_num_workers(M),
        display_worker_counts(0, M).

display_worker_counts(M, M) :- !.
display_worker_counts(W, M) :-
        get_counter(W, C),
        (   C==0 -> true ;
            format('Solutions by worker ~d: ~d ~n', [W,C])
        ),
        W1 is W+1, display_worker_counts(W1, M).
```

This example is compiled and executed as follows to count the number of solutions to the 8 queens problem (see Section 3.2 [Run Example], page 41):

```
% gcc -c count.c
% sicstus -P
| ?- compile([count,queens]).
| ?- muse_flag(num_workers,_,3).
| ?- count(queens(8,_)).
Solutions by worker 0: 39
Solutions by worker 1: 31
Solutions by worker 2: 22
Total 92 solutions in 0.070 seconds.

yes
| ?-
```

11 The Prolog Library

The Prolog library comprises a number of packages which are thought to be useful in a number of applications. Note that the predicates in the Prolog library are not built-in predicates. One has to explicitly load each package to get access to its predicates. The following packages are provided:

`arrays` provides an implementation of extendible arrays with logarithmic access time.

`assoc` uses AVL trees to implement "association lists", i.e. extendible finite mappings from terms to terms.

`atts` provides a means of associating with variables arbitrary attributes, i.e. named properties that can be used as storage locations as well as hooks into Prolog's unification.

`heaps` implements binary heaps, the main application of which are priority queues.

`lists` provides basic operations on lists.

`terms` provides a number of operations on terms.

`ordsets` defines operations on sets represented as lists with the elements ordered in Prolog standard order.

`queues` defines operations on queues (FIFO stores of information).

`random` provides a random number generator.

`system` provides access to operating system services.

`trees` uses binary trees to represent non-extendible arrays with logarithmic access time. The functionality is very similar to that of `library(arrays)`, but `library(trees)` is slightly more efficient if the array does not need to be extendible.

`ugraphs` provides an implementation of directed and undirected graphs with unlabeled edges.

`wgraphs` provides an implementation of directed and undirected graphs where each edge has an integral weight.

`sockets` provides an interface to system calls for manipulating sockets.

`linda/client`
`linda/server`
provides an implementation of the Linda concept for process communication.

`db` provides storage and retrieval of terms on disk files with user-defined multiple indexing.

`clpb` provides constraint solving over Booleans.

`clpq`
`clpr` provides constraint solving over Q (Rationals) or R (Reals).

`clpfd` provides constraint solving over Finite (Integer) Domains

`chr` provides Constraint Handling Rules

`objects` provides the combination of the logic programming and the object-oriented programming paradigms.

`gcla` is a specification tool that is based on Generalized Horn Clause Language, a generalization of Prolog.

`tcltk` An interface to the *Tcl/Tk* language and toolkit.

`gauge` is a profiling tool for Prolog programs with a graphical interface based on `tcltk`.

`charsio` defines I/O predicates that read from, or write to, a list of character codes.

`jasper` An interface to the Java language.

`flinkage` is a utility program for generating glue code for the Foreign Language Interface when building statically linked Runtime Systems or Development Systems.

`timeout` provides a way of running goals with an execution time limit.

`xref` provides a cross reference producer for debugging and program analysis.

To load a library package *Package*, you will normally enter a query

```
| ?- use_module(library(Package)).
```

A library package normally consists of one or more hidden modules.

An alternative way of loading from the library is using the built-in predicate `require/1` (see Section 7.1.1 [Read In], page 73). The index file '`INDEX.pl`' needed by `require/1` can be created by the `make_index` program. This program is loaded as:

```
| ?- use_module(library(mkindex)).
```

`make_index:make_library_index(+`*LibraryDirectory*`)`
: Creates a file '`INDEX.pl`' in *LibraryDirectory*. All '`*.pl`' files in the directory and all its subdirectories are scanned for `module/2` declarations. From these declarations, the exported predicates are entered into the index.

12 Array Operations

This package provides an implementation of extendible arrays with logarithmic access time.

Beware: the atom `$` is used to indicate an unset element, and the functor `$ /4` is used to indicate a subtree. In general, array elements whose principal function symbol is `$` will not work.

To load the package, enter the query

```
| ?- use_module(library(arrays)).
```

`new_array(-`*Array*`)`
: Binds *Array* to a new empty array. Example:

```
| ?- new_array(A).

A = array($($,$,$,$),2) ?

yes
```

`is_array(+`*Array*`)`
: Is true when *Array* actually is an array.

`aref(+`*Index*`, +`*Array*`, ?`*Element*`)`
: *Element* is the element at position *Index* in *Array*. It fails if *Array*`[`*Index*`]` is undefined.

`arefa(+`*Index*`, +`*Array*`, ?`*Element*`)`
: Is like `aref/3` except that *Element* is a new array if *Array*`[`*Index*`]` is undefined. Example:

```
| ?- arefa(3, array($($,$,$,$),2), E).

E = array($($,$,$,$),2) ?

yes
```

`arefl(+`*Index*`, +`*Array*`, ?`*Element*`)`
: Is as `aref/3` except that *Element* is `[]` for undefined cells. Example:

```
| ?- arefl(3, array($($,$,$,$),2), E).

E = [] ?

yes
```

`array_to_list(+`*Array*`, -`*List*`)`
: *List* is a list with the pairs *Index*-*Element* of all the elements of *Array*. Example:

```
| ?- array_to_list(array($(a,b,c,d),2), List).

List = [0-a,1-b,2-c,3-d] ?

yes
```

`aset(+`*Index*`, +`*Array*`, +`*Element*`, -`*NewArray*`)`

NewArray is the result of setting *Array*`[`*Index*`]` to *Element*. Example:

```
| ?- aset(3,array($($,$,$,$),2), a, Newarr).

Newarr = array($($,$,$,a),2) ?

yes
```

13 Association Lists

In this package, finite mappings ("association lists") are represented by AVL trees, i.e. they are subject to the Adelson-Velskii-Landis balance criterion:

> A tree is balanced iff for every node the heights of its two subtrees differ by at most 1.

The empty tree is represented as `t`. A tree with key `K`, value `V`, and left and right subtrees `L` and `R` is represented as `t(K,V,|R|-|L|,L,R)`, where `|T|` denotes the height of `T`.

The advantage of this representation is that lookup, insertion and deletion all become—in the worst case—*O(log n)* operations.

The algorithms are from [Wirth 76], section 4.4.6–4.4.8.

To load the package, enter the query

```
| ?- use_module(library(assoc)).
```

`empty_assoc(`*?Assoc*`)`
: *Assoc* is an empty AVL tree.

`assoc_to_list(+`*Assoc*`, `*?List*`)`
: *List* is a list of *Key-Value* pairs in ascending order with no duplicate *Keys* specifying the same finite function as the association tree *Assoc*. Use this to convert an association tree to a list.

`is_assoc(+`*Assoc*`)`
: *Assoc* is a (proper) AVL tree. It checks both that the keys are in ascending order and that *Assoc* is properly balanced.

`min_assoc(+`*Assoc*`, `*?Key*`, `*?Val*`)`
: *Key* is the smallest key in *Assoc* and *Val* is its value.

`max_assoc(+`*Assoc*`, `*?Key*`, `*?Val*`)`
: *Key* is the greatest key in *Assoc* and *Val* is its value.

`gen_assoc(`*?Key*`, +`*Assoc*`, `*?Value*`)`
: *Key* is associated with *Value* in the association tree *Assoc*. Can be used to enumerate all *Values* by ascending *Keys*.

`get_assoc(+`*Key*`, +`*Assoc*`, `*?Value*`)`
: *Key* is identical (==) to one of the keys in the association tree *Assoc*, and *Value* unifies with the associated value.

`get_assoc(+`*Key*`, +`*OldAssoc*`, `*?OldValue*`, `*?NewAssoc*`, `*?NewValue*`)`
: *OldAssoc* and *NewAssoc* are association trees of the same shape having the same elements except that the value for *Key* in *OldAssoc* is *OldValue* and the value for *Key* in *NewAssoc* is *NewValue*.

`get_next_assoc(+`*Key*`, +`*Assoc*`, `*?Knext*`, `*?Vnext*`)`
: *Knext* and *Vnext* is the next key and associated value after *Key* in *Assoc*.

`get_prev_assoc(+`*Key*`, +`*Assoc*`, ?`*Kprev*`, ?`*Vprev*`)`
: *Kprev* and *Vprev* is the previous key and associated value after *Key* in *Assoc*.

`list_to_assoc(+`*List*`, ?`*Assoc*`)`
: *List* is a proper list of *Key-Value* pairs (in any order) and *Assoc* is an association tree specifying the same finite function from *Keys* to *Values*.

`ord_list_to_assoc(+`*List*`, ?`*Assoc*`)`
: *List* is a proper list of *Key-Value* pairs (keysorted) and *Assoc* is an association tree specifying the same finite function from *Keys* to *Values*.

`map_assoc(:`*Pred*`, ?`*Assoc*`)`
: *Assoc* is an association tree, and for each *Key*, if *Key* is associated with *Value* in *Assoc*, *Pred(Value)* is true.

`map_assoc(:`*Pred*`, ?`*OldAssoc*`, ?`*NewAssoc*`)`
: *OldAssoc* and *NewAssoc* are association trees of the same shape, and for each *Key*, if *Key* is associated with *Old* in *OldAssoc* and with *New* in *NewAssoc*, *Pred(Old,New)* is true.

`put_assoc(+`*Key*`, +`*OldAssoc*`, +`*Val*`, ?`*NewAssoc*`)`
: *OldAssoc* and *NewAssoc* define the same finite function, except that *NewAssoc* associates Val with *Key*. *OldAssoc* need not have associated any value at all with *Key*.

`del_assoc(+`*Key*`, +`*OldAssoc*`, ?`*Val*`, ?`*NewAssoc*`)`
: *OldAssoc* and *NewAssoc* define the same finite function except that *OldAssoc* associates *Key* with *Val* and *NewAssoc* doesn't associate *Key* with any value.

`del_min_assoc(+`*OldAssoc*`, ?`*Key*`, ?`*Val*`, ?`*NewAssoc*`)`
: *OldAssoc* and *NewAssoc* define the same finite function except that *OldAssoc* associates *Key* with *Val* and *NewAssoc* doesn't associate *Key* with any value and *Key* precedes all other keys in *OldAssoc*.

`del_max_assoc(+`*OldAssoc*`, ?`*Key*`, ?`*Val*`, -`*NewAssoc*`)`
: *OldAssoc* and *NewAssoc* define the same finite function except that *OldAssoc* associates *Key* with *Val* and *NewAssoc* doesn't associate *Key* with any value and *Key* is preceded by all other keys in *OldAssoc*.

14 Attributed Variables

This package implements attributed variables. It provides a means of associating with variables arbitrary attributes, i.e. named properties that can be used as storage locations as well as to extend the default unification algorithm when such variables are unified with other terms or with each other. This facility was primarily designed as a clean interface between Prolog and constraint solvers, but has a number of other uses as well. The basic idea is due to Christian Holzbaur and he was actively involved in the final design. For background material, see the dissertation [Holzbaur 90].

To load the package, enter the query

```
| ?- use_module(library(atts)).
```

The package provides a means to declare and access named attributes of variables. The attributes are compound terms whose arguments are the actual attribute values. The attribute names are *private* to the module in which they are defined. They are defined with a declaration

`:- attribute` *AttributeSpec*`, ..., `*AttributeSpec*`.`

where each *AttributeSpec* has the form (*Name*/*Arity*). There must be at most one such declaration in a module *Module*.

Having declared some attribute names, these attributes can now be added, updated and deleted from unbound variables. For each declared attribute name, any variable can have at most one such attribute (initially it has none).

The declaration causes the following two access predicates to become defined by means of the `user:goal_expansion/3` mechanism. They take a variable and an *AccessSpec* as arguments where an *AccessSpec* is either `+(`*Attribute*`)`, `-(`*Attribute*`)`, or a list of such. The `+` prefix may be dropped for convenience. The meaning of the `+/-` prefix is documented below:

Module`:get_atts(-`*Var*`, ?`*AccessSpec*`)`
: Gets the attributes of *Var* according to *AccessSpec*. If *AccessSpec* is unbound, it will be bound to a list of all set attributes of *Var*. Non-variable terms cause a type error to be raised. The prefixes in the *AccessSpec* have the following meaning:

  `+(`*Attribute*`)`
  : The corresponding actual attribute must be present and is unified with *Attribute*.

  `-(`*Attribute*`)`
  : The corresponding actual attribute must be absent. The arguments of *Attribute* are ignored, only the name and arity are relevant.

Module`:put_atts(-`*Var*`, +`*AccessSpec*`)`
: Sets the attributes of *Var* according to *AccessSpec*. Non-variable terms cause a type error to be raised. The effects of `put_atts/2` are undone on backtracking.

`+(`*Attribute*`)`
: The corresponding actual attribute is set to *Attribute*. If the actual attribute was already present, it is simply replaced.

`-(`*Attribute*`)`
: The corresponding actual attribute is removed. If the actual attribute was already absent, nothing happens.

A module that contains an attribute declaration has an opportunity to extend the default unification algorithm by defining the following predicate:

Module`:verify_attributes(-`*Var*`, +`*Value*`, -`*Goals*`)`
: *A hook predicate.* This predicate is called whenever a variable *Var* that might have attributes in *Module* is about to be bound to *Value* (it might have none). The unification resumes after the call to `verify_attributes/3`. *Value* is a non-variable term, or another attributed variable. *Var* might have no attributes present in *Module*; the unification extension mechanism is not sophisticated enough to filter out exactly the variables that are relevant for *Module*.

  `verify_attributes/3` is called *before* *Var* has actually been bound to *Value*. If it fails, the unification is deemed to have failed. It may succeed non-deterministically, in which case the unification might backtrack to give another answer. It is expected to return, in *Goals*, a list of goals to be called *after* *Var* has been bound to *Value*.

  `verify_attributes/3` may invoke arbitrary Prolog goals, but *Var* should *not* be bound by it. Binding *Var* will result in undefined behavior.

  If *Value* is a non-variable term, `verify_attributes/3` will typically inspect the attributes of *Var* and check that they are compatible with *Value* and fail otherwise. If *Value* is another attributed variable, `verify_attributes/3` will typically copy the attributes of *Var* over to *Value*, or merge them with *Value*'s, in preparation for *Var* to be bound to *Value*. In either case, `verify_attributes/3` may determine *Var*'s current attributes by calling `get_atts(`*Var*`,`*List*`)` with an unbound *List*.

An important use for attributed variables is in implementing coroutining facilities as an alternative or complement to the built-in coroutining mechanisms. In this context it might be useful to be able to interpret some of the attributes of a variable as a goal that is blocked on that variable. Certain built-in predicates (`frozen/2`, `call_residue/2`) and the Prolog top level need to access blocked goals, and so need a means of getting the goal interpretation of attributed variables by calling:

Module`:attribute_goal(-`*Var*`, -`*Goal*`)`
: *A hook predicate.* This predicate is called in each module that contains an attribute declaration, when an interpretation of the attributes as a goal is needed. It should unify *Goal* with the interpretation, or merely fail if no such interpretation is available.

An important use for attributed variables is to provide an interface to constraint solvers. An important function for a constraint solver in the constraint logic programming paradigm is to be able to perform projection of the residual constraints onto the variables that occurred in the top-level query. A module that contains an attribute declaration has an opportunity to perform such projection of its residual constraints by defining the following predicate:

Module:`project_attributes(+`*QueryVars*`, +`*AttrVars*`)`

A hook predicate. This predicate is called by the Prolog top level and by the built-in predicate `call_residue/2` in each module that contains an attribute declaration. *QueryVars* is the list of variables occurring in the query, or in terms bound to such variables, and *AttrVars* is a list of possibly attributed variables created during the execution of the query. The two lists of variables may or may not be disjoint.

If the attributes on *AttrVars* can be interpreted as constraints, this predicate will typically "project" those constraints onto the relevant *QueryVars*. Ideally, the residual constraints will be expressed entirely in terms of the *QueryVars*, treating all other variables as existentially quantified. Operationally, `project_attributes/2` must remove all attributes from *AttrVars*, and add transformed attributes representing the projected constraints to some of the *QueryVars*.

Projection has the following effect on the Prolog top level. When the top level query has succeeded, `project_attributes/2` is called first. The top level then prints the answer substition and residual constraints. While doing so, it searches for attributed variables created during the execution of the query. For each such variable, it calls `attribute_goal/2` to get a printable representation of the constraint encoded by the attribute. Thus, `project_attributes/2` is a mechanism for controlling how the residual constraints should be displayed at top level.

Similarly during the execution of `call_residue(`*Goal*`,`*Residue*`)`, when *Goal* has succeeded, `project_attributes/2` is called. After that, all attributed variables created during the execution of *Goal* are located. For each such variable, `attribute_goal/2` produces a term representing the constraint encoded by the attribute, and *Residue* is unified with the list of all such terms.

The exact definition of `project_attributes/2` is constraint system dependent, but see Section 29.5 [Projection], page 259 for details about projection in clp(Q,R).

In the following example we sketch the implementation of a finite domain "solver". Note that an industrial strength solver would have to provide a wider range of functionality and that it quite likely would utilize a more efficient representation for the domains proper. The module exports a single predicate `domain(-`*Var*`,?`*Domain*`)` which associates *Domain* (a list of terms) with *Var*. A variable can be queried for its domain by leaving *Domain* unbound.

We do not present here a definition for `project_attributes/2`. Projecting finite domain constraints happens to be difficult.

```
:- module(domain, [domain/2]).

:- use_module(library(atts)).
:- use_module(library(ordsets), [
        ord_intersection/3,
        ord_intersect/2,
        list_to_ord_set/2
   ]).

:- attribute dom/1.
```

```
verify_attributes(Var, Other, Goals) :-
        get_atts(Var, dom(Da)), !,          % are we involved?
        (   var(Other) ->                   % must be attributed then
            (   get_atts(Other, dom(Db)) -> %   has a domain?
                ord_intersection(Da, Db, Dc),
                Dc = [El|Els],              % at least one element
                (   Els = [] ->             % exactly one element
                    Goals = [Other=El]      % implied binding
                ;   Goals = [],
                    put_atts(Other, dom(Dc))% rescue intersection
                )
            ;   Goals = [],
                put_atts(Other, dom(Da))    % rescue the domain
            )
        ;   Goals = [],
            ord_intersect([Other], Da)      % value in domain?
        ).
verify_attributes(_, _, []).                % unification triggered
                                            % because of attributes
                                            % in other modules

attribute_goal(Var, domain(Var,Dom)) :-     % interpretation as goal
        get_atts(Var, dom(Dom)).

domain(X, Dom) :-
        var(Dom), !,
        get_atts(X, dom(Dom)).
domain(X, List) :-
        list_to_ord_set(List, Set),
        Set = [El|Els],                     % at least one element
        (   Els = [] ->                     % exactly one element
            X = El                          % implied binding
        ;   put_atts(Fresh, dom(Set)),
            X = Fresh                       % may call
                                            % verify_attributes/3
        ).
```

Note that the "implied binding" `Other=El` was deferred until after the completion of `verify_attribute/3`. Otherwise, there might be a danger of recursively invoke `verify_attribute/3`, which might bind `Var`, which is not allowed inside the scope of `verify_attribute/3`. Deferring unifications into the third argument of `verify_attribute/3` effectively serializes th calls to `verify_attribute/3`.

Assuming that the code resides in the file '`domain.pl`', we can use it via:

```
| ?- use_module(domain).
```

Let's test it:

```
| ?- domain(X,[5,6,7,1]), domain(Y,[3,4,5,6]), domain(Z,[1,6,7,8]).

domain(X,[1,5,6,7]),
domain(Y,[3,4,5,6]),
domain(Z,[1,6,7,8]) ?

yes
| ?- domain(X,[5,6,7,1]), domain(Y,[3,4,5,6]), domain(Z,[1,6,7,8]),
     X=Y.

Y = X,
domain(X,[5,6]),
domain(Z,[1,6,7,8]) ?

yes
| ?- domain(X,[5,6,7,1]), domain(Y,[3,4,5,6]), domain(Z,[1,6,7,8]),
     X=Y, Y=Z.

X = 6,
Y = 6,
Z = 6
```

To demonstrate the use of the *Goals* argument of `verify_attributes/3`, we give an implementation of `freeze/2`. We have to name it `myfreeze/2` in order to avoid a name clash with the built-in predicate of the same name.

```
:- module(myfreeze, [myfreeze/2]).

:- use_module(library(atts)).

:- attribute frozen/1.

verify_attributes(Var, Other, Goals) :-
        get_atts(Var, frozen(Fa)), !,       % are we involved?
        (   var(Other) ->                   % must be attributed then
            (   get_atts(Other, frozen(Fb)) % has a pending goal?
            ->  put_atts(Other, frozen((Fa,Fb))) % rescue conjunction
            ;   put_atts(Other, frozen(Fa)) % rescue the pending goal
            ),
            Goals = []
        ;   Goals = [Fa]
        ).
verify_attributes(_, _, []).

attribute_goal(Var, Goal) :-                % interpretation as goal
        get_atts(Var, frozen(Goal)).

myfreeze(X, Goal) :-
        put_atts(Fresh, frozen(Goal)),
```

```
        Fresh = X.
```

Assuming that this code lives in file '`myfreeze.pl`', we would use it via:

```
| ?- use_module(myfreeze).
| ?- myfreeze(X,print(bound(x,X))), X=2.

bound(x,2)                      % side effect
X = 2                           % bindings
```

The two solvers even work together:

```
| ?- myfreeze(X,print(bound(x,X))), domain(X,[1,2,3]),
     domain(Y,[2,10]), X=Y.

bound(x,2)                      % side effect
X = 2,                          % bindings
Y = 2
```

The two example solvers interact via bindings to shared attributed variables only. More complicated interactions are likely to be found in more sophisticated solvers. The corresponding `verify_attributes/3` predicates would typically refer to the attributes from other known solvers/modules via the module prefix in *Module*`:get_atts/2`.

15 Heap Operations

A binary heap is a tree with keys and associated values that satisfies the *heap codition*: the key of every node is greater than or equal to the key of its parent, if it has one. The main application of binary heaps are priority queues. To load the package, enter the query

```
| ?- use_module(library(heaps)).
```

`add_to_heap(+`*OldHeap*`, +`*Key*`, +`*Datum*`, ?`*NewHeap*`)`
: Inserts the new *Key-Datum* pair into the current heap *OldHeap* producing the new heap *NewHeap*. The insertion is not stable, that is, if you insert several pairs with the same *Key* it is not defined which of them will come out first, and it is possible for any of them to come out first depending on the history of the heap. Example:

```
| ?- add_to_heap(t(0,[],t),3,678,N).

N = t(1,[],t(3,678,t,t)) ?

yes
```

`get_from_heap(+`*OldHeap*`, ?`*Key*`, ?`*Datum*`, ?`*NewHeap*`)`
: Returns the *Key-Datum* pair in *OldHeap* with the smallest *Key*, and also a *NewHeap* which is the *OldHeap* with that pair deleted. Example:

```
get_from_heap(t(1,[],t(1,543,t,t)),K,D,N).

D = 543,
K = 1,
N = t(0,[1],t) ?

yes
```

`empty_heap(?`*Heap*`)`
: is true when *Heap* is the empty heap.

`heap_size(+`*Heap*`, ?`*Size*`)`
: *Size* is the number of elements in the heap *Heap*.

`heap_to_list(+`*Heap*`, -`*List*`)`
: Returns the current set of *Key-Datum* pairs in the *Heap* as a keysorted *List*.

`is_heap(+`*Heap*`)`
: is true when *Heap* is a valid heap.

`list_to_heap(+`*List*`, -`*Heap*`)`
: Takes a list *List* of *Key-Datum* pairs and forms them into a heap *Heap*. Example:

```
| ?- list_to_heap([1-34,2-345,5-678],H).

H = t(3,[],t(1,34,t(2,345,t,t),t(5,678,t,t))) ?

yes
```

`min_of_heap(+`*Heap*`, ?`*Key*`, ?`*Datum*`)`
: Returns the *Key*-*Datum* pair at the top of the heap *Heap* without removing it. Fails if the heap is empty.

`min_of_heap(+`*Heap*`, ?`*Key1*`, ?`*Datum1*`, ?`*Key2*`, ?`*Datum2*`)`
: Returns the smallest (*Key1*-*Datum1*) and second smallest (*Key2*-*Datum2*) pairs in the *Heap*, without deleting them. It fails if the heap does not have at least two elements.

`delete_from_heap(+`*OldHeap*`, +`*Key*`, ?`*Datum*`, ?`*NewHeap*`)`
: deletes a single *Key*-*Datum* pair in *OldHeap* producing *NewHeap*. This is useful if you want to e.g. change the priority of *Datum*. Beware: this operation needs to search the whole heap in the worst case.

16 List Operations

This package defines operations on lists. Lists are a very basic data structure, but nevertheless certain very frequent operations are provided in this package.

To load the package, enter the query

```
| ?- use_module(library(lists)).
```

`append(`*?Prefix*`, `*?Suffix*`, `*?Combined*`)`
: *Combined* is the combined list of the elements in *Prefix* followed by the elements in *Suffix*. It can be used to form *Combined* or it can be used to find *Prefix* and/or *Suffix* from a given *Combined*.

`delete(+`*List*`, +`*Element*`, `*?Residue*`)`
: *Residue* is the result of removing all *identical* occurrences of *Element* in *List*.

`is_list(+`*List*`)`
: *List* is a proper list.

`last(`*?List*`, `*?Last*`)`
: *Last* is the last element in *List*. Example:

```
| ?- last([x,y,z], Z).

Z = z ?

yes
```

`max_list(+`*ListOfNumbers*`, `*?Max*`)`
: *Max* is the largest of the elements in *ListOfNumbers*.

`member(`*?Element*`, `*?List*`)`
: *Element* is a member of *List*. It may be used to test for membership in a list, but it can also be used to enumerate all the elements in *List*. Example:

```
| ?- member(X, [a,b,c]).

X = a ? ;

X = b ? ;

X = c ?

yes
```

`memberchk(+`*Element*`, +`*List*`)`
: *Element* is a member of *List*, but `memberchk/2` only succeeds once and can therefore not be used to enumerate the elements in *List*. Example:

```
| ?- memberchk(X, [a,b,c]).

X = a ? ;

no
```

`min_list(+`*ListOfNumbers*`, ?`*Min*`)`
: *Min* is the smallest of the numbers in the list *ListOfNumbers*.

`nextto(?`*X*`, ?`*Y*`, ?`*List*`)`
: *X* and *Y* appears side-by-side in *List*. Example:

```
| ?- nextto(X, Y, [1,2,3]).

X = 1,
Y = 2 ? ;

X = 2,
Y = 3 ? ;

no
```

`no_doubles(?`*List*`)`
: *List* contains no duplicated elements. This is true when `dif(`*X*`, `*Y*`)` holds for all pairs of members *X* and *Y* of the list.

`non_member(?`*Element*`, ?`*List*`)`
: *Element* does not occur in *List*. This is true when `dif(`*Element*`, `*Y*`)` holds for all members *Y* of the list.

`nth(?`*N*`, ?`*List*`, ?`*Element*`)`
: *Element* is the *N*th element of *List*. The first element is number 1. Example:

```
| ?- nth(N, [a,b,c,d,e,f,g,h,i], f).

N = 6 ?

yes
```

`nth(?`*N*`, ?`*List*`, ?`*Element*`, ?`*Rest*`)`
: *Element* is in position *N* in the *List* and *Rest* is all elements in *List* except *Element*.

`nth0(?`*N*`, ?`*List*`, ?`*Element*`)`
: *Element* is the *N*th element of *List*, counting the first element as 0.

`nth0(?`*N*`, ?`*List*`, ?`*Element*`, ?`*Rest*`)`
: *Element* is the *N*th element of *List*, counting the first element as 0. *Rest* is all the other elements in *List*. Example:

```
| ?- nth0(N, [a,b,c,d,e,f,g,h,i,j], f, R).

N = 5,
R = [a,b,c,d,e,g,h,i,j] ?

yes
```

`permutation(`*?List*`, `*?Perm*`)`
: *Perm* is a permutation of *List*.

`prefix(`*?Prefix*`, `*?List*`)`
: *Prefix* is a prefix of *List*. Example:

```
| ?- prefix([1,2,3], [1,2,3,4,5,6]).

yes
```

`remove_duplicates(+`*List*`, `*?Pruned*`)`
: *Pruned* is the result of removing all *identical* duplicate elements in *List*. Example:

```
| ?- remove_duplicates([1,2,3,2,3,1], P).

P = [1,2,3] ? ;

no
```

`reverse(`*?List*`, `*?Reversed*`)`
: *Reversed* has the same elements as *List* but in a reversed order.

`same_length(`*?List1*`, `*?List2*`)`
: *List1* and *List2* have the same number of elements.

`same_length(`*?List1*`, `*?List2*`, `*?Length*`)`
: *List1* and *List2* have the same number of elements and that number is *Length*. Example:

```
| ?- same_length([1,2,3], [9,8,7], N).

N = 3 ? ;

no
```

`select(`*?Element*`, `*?List*`, `*?List2*`)`
: The result of removing an occurrence of *Element* in *List* is *List2*.

`sublist(`*?Sub*`, `*?List*`)`
: *Sub* contains some of the elements of *List*, in the same order.

`substitute(+`*X*`, +`*Xlist*`, +`*Y*`, `*?Ylist*`)`
: *Xlist* and *Ylist* are equal except for replacing *identical* occurrences of *X* by *Y*. Example:

```
| ?- substitute(1, [1,2,3,4], 5, X).

X = [5,2,3,4] ?

yes
```

`sum_list(+`*ListOfNumbers*`, `*?Sum*`)`
: *Sum* is the result of adding the *ListOfNumbers* together.

`suffix(`*?Suffix*`, `*?List*`)`
: *Suffix* is a suffix of *List*.

17 Term Utilities

This package defines operations on terms for subsumption checking, "anti-unification", unification with occurs-check, testing acyclicity, and getting the variables.

To load the package, enter the query

```
| ?- use_module(library(terms)).
```

`subsumes_chk(`*?General*`, `*?Specific*`)`
: *Specific* is an instance of *General*, i.e. if there is a substitution that leaves *Specific* unchanged and makes *General* identical to *Specific*. It doesn't bind any variables.

```
subsumes_chk(f(X), f(a)).

true

| ?- subsumes_chk(f(a), f(X)).

no

| ?- subsumes_chk(A-A, B-C).

no

| ?- subsumes_chk(A-B, C-C).

true
```

`subsumes(`*?General*`, `*?Specific*`)`
: *Specific* is an instance of *General*. It will bind variables in *General* (but not those in *Specific*) so that *General* becomes identical to *Specific*.

`variant(`*?Term*`, `*?Variant*`)`
: *Term* and *Variant* are identical modulo renaming of variables, provided *Term* and *Variant* have no variables in common.

`term_subsumer(`*?Term1*`, `*?Term2*`, `*?General*`)`
: *General* is the most specific term that generalizes *Term1* and *Term2*. This process is sometimes called *anti-unification*, as it is the dual of unification.

```
| ?- term_subsumer(f(g(1,h(_))), f(g(_,h(1))), T).

T = f(g(_B,h(_A)))

| ?- term_subsumer(f(1+2,2+1), f(3+4,4+3), T).

T = f(_A+_B,_B+_A)
```

`term_hash(`*?Term*`, `*?Hash*`)`
`term_hash(`*?Term*`, +`*Depth*`, +`*Range*`, `*?Hash*`)`

If *Term* is instantiated up to the given *Depth*, an integer hash value in the range [0,*Range*) as a function of *Term* is unified with *Hash*. Otherwise, the goal just succeeds, leaving *Hash* uninstantiated.

If *Term* contains floats or integers outside the range `[-33554432,33554431]`, the hash value will be platform dependent. Otherwise, the hash value will be identical across runs and platforms.

The *depth* of a term is defined as follows: the (principal functor of) the term itself has depth 1, and an argument of a term with depth *i* has depth *i+1*.

Depth should be an integer `>=` -1. If *Depth* = -1 (the default), *Term* must be ground, and all subterms of *Term* are relevant in computing *Hash*. Otherwise, only the subterms up to depth *Depth* of *Term* are used in the computation.

Range should be an integer `>=` 1. The default will give hash values in a range appropriate for all platforms.

```
| ?- term_hash([a,b,_], 3, 4, H).

H = 2

| ?- term_hash([a,b,_], 4, 4, H).

true

| ?- term_hash(f(a,f(b,f(_,[]))), 2, 4, H).

H = 2
```

`term_hash/(2,4)` is provided primarily as a tool for the construction of sophisticated Prolog clause access schemes. Its intended use is to generate hash values for terms that will be used with first argument clause indexing, yielding compact and efficient multi-argument or deep argument indexing.

`term_variables(`*?Term*`, `*?Variables*`)`

Variables is the set of variables occurring in *Term*.

`unify_with_occurs_check(`*?X*`, `*?Y*`)` *[ISO]*

True if X and Y unify to a finite (acyclic) term. Runs in almost linear time.

`acyclic_term(`*?X*`)`

True if X is finite (acyclic). Runs in linear time.

`cyclic_term(`*?X*`)`

True if X is infinite (cyclic). Runs in linear time.

18 Ordered Set Operations

This package defines operations on ordered sets. Ordered sets are sets represented as lists with the elements ordered in a standard order. The ordering is defined by the `@<` family of term comparison predicates and it is the ordering produced by the built-in predicate `sort/2` (see Section 7.3 [Term Compare], page 97).

To load the package, enter the query

```
| ?- use_module(library(ordsets)).
```

`is_ordset(+`*Set*`)`
: *Set* is an ordered set.

`list_to_ord_set(+`*List*`, ?`*Set*`)`
: *Set* is the ordered representation of the set denoted by the unordered representation *List*. Example:

```
| ?- list_to_ord_set([p,r,o,l,o,g], P).

P = [g,l,o,p,r] ?

yes
```

`ord_add_element(+`*Set1*`, +`*Element*` ?`*Set2*`)`
: *Set2* is *Set1* with *Element* inserted in it, preserving the order. Example:

```
| ?- ord_add_element([a,c,d,e,f], b, N).

N = [a,b,c,d,e,f] ?

yes
```

`ord_del_element(+`*Set1*`, +`*Element*`, ?`*Set2*`)`
: *Set2* is like *Set1* but with *Element* removed.

`ord_disjoint(+`*Set1*`, +`*Set2*`)`
: The two ordered sets have no elements in common.

`ord_intersect(+`*Set1*`, +`*Set2*`)`
: The two ordered sets have at least one element in common.

`ord_intersection(+`*Set1*`, +`*Set2*`, ?`*Intersect*`)`
: *Intersect* is the ordered set representation of the intersection between *Set1* and *Set2*.

`ord_intersection(+`*Set1*`, +`*Set2*`, ?`*Intersect*`, ?`*Diff*`)`
: *Intersect* is the intersection between *Set1* and *Set2*, and *Diff* is the difference between *Set2* and *Set1*.

`ord_intersection(+`*Sets*`, ?`*Intersection*`)`
: *Intersection* is the ordered set representation of the intersection of all the sets in *Sets*. Example:

```
| ?- ord_intersection([[1,2,3],[2,3,4],[3,4,5]], I).

I = [3] ?

yes
```

`ord_member(+`*Elt*`, +`*Set*`)`
: is true when *Elt* is a member of *Set*.

`ord_seteq(+`*Set1*`, +`*Set2*`)`
: Is true when the two arguments represent the same set. Since they are assumed to be ordered representations, they must be identical.

`ord_setproduct(+`*Set1*`, +`*Set2*`, ?`*SetProduct*`)`
: *SetProduct* is the Cartesian Product of the two Sets. The product is represented as pairs: Elem1-Elem2 where Elem1 is an element from *Set1* and Elem2 is an element from *Set2*. Example

```
| ?- ord_setproduct([1,2,3], [4,5,6], P).

P = [1-4,1-5,1-6,2-4,2-5,2-6,3-4,3-5,3-6] ?

yes
```

`ord_subset(+`*Set1*`, +`*Set2*`)`
: Every element of the ordered set *Set1* appears in the ordered set *Set2*.

`ord_subtract(+`*Set1*`, +`*Set2*`, ?`*Difference*`)`
: *Difference* contains all and only the elements of *Set1* which are not also in *Set2*. Example:

```
| ?- ord_subtract([1,2,3,4], [3,4,5,6], S).

S = [1,2] ?

yes
```

`ord_symdiff(+`*Set1*`, +`*Set2*`, ?`*Difference*`)`
: *Difference* is the symmetric difference of *Set1* and *Set2*. Example:

```
| ?- ord_symdiff([1,2,3,4], [3,4,5,6], D).

D = [1,2,5,6] ?

yes
```

`ord_union(+`*Set1*`, +`*Set2*`, ?`*Union*`)`
: *Union* is the union of *Set1* and *Set2*.

`ord_union(+`*Set1*`, +`*Set2*`, ?`*Union*`, ?`*New*`)`
: *Union* is the union of *Set1* and *Set2*, and *New* is the difference between *Set2* and *Set1*. This is useful if you are accumulating members of a set and you want to process new elements as they are added to the set.

`ord_union(+`*Sets*`, ?`*Union*`)`

Union is the union of all the sets in *Sets*. Example:

```
| ?- ord_union([[1,2,3],[2,3,4],[3,4,5]], U).

U = [1,2,3,4,5] ?

yes
```

19 Queue Operations

A queue is a first-in, first-out store of information. This implementation of queues uses difference-lists, the head of the difference-list represents the beginning of the queue and the tail represents the end of the queue. The members of the difference-list are the elements in the queue. The first argument in the queue-representation is the number of elements in the queue in unary representation.

Thus, a queue with *n* elements is represented as follows:

```
q(s(...s(0)...), [X1,...,Xn,Y1,...,Ym], [Y1,...,Ym])
```

where *n* is the length of the queue and *X1...Xn* are the elements of the queue.

To load the package, enter the query

```
| ?- use_module(library(queues)).
```

`empty_queue(`*?Queue*`)`
: Is true if *Queue* has no elements.

`is_queue(+`*Queue*`)`
: is true when *Queue* is a valid queue.

`queue(`*?X*`, `*?Queue*`)`
: Is true if *Queue* has one element and that is *X*.

`queue_head(`*?Head*`, `*?Queue1*`, `*?Queue2*`)`
: *Queue1* and *Queue2* are the same queues except that *Queue2* has *Head* inserted in the front. It can be used to enqueue the first element in *Queue2*. Example:

```
| ?- queue_head(Head, Nq,
                 q(s(s(s(s(0)))),[1,2,3,4|R],R)).

Head = 1,
Nq = q(s(s(s(0))),[2,3,4|_193],_193),
R = _193 ?

yes
```

`queue_head_list(+`*HeadList*`, `*?Queue1*`, `*?Queue2*`)`
: *Queue1* and *Queue2* have the same elements except that *Queue2* has *HeadList* inserted in the front.

`queue_last(`*?Last*`, `*?Queue1*`, `*?Queue2*`)`
: *Queue2* is like *Queue1* but have *Last* as the last element in the queue.

`queue_last_list(+`*LastList*`, `*?Queue1*`, `*?Queue2*`)`
: *Queue1* and *Queue2* are the same queues except that *Queue2* has the list of elements *LastList* last in the queue. Example:

```
| ?- queue_last_list([5,6], q(s(s(0))), [1,2|R], R), NQ).

NQ = q(s(s(s(s(0))))),[1,2,5,6|_360],_360),
R = [5,6|_360] ?

yes
```

`list_queue(+`*List*`, ?`*Queue*`)`

Queue is the queue representation of the elements in *List*. Example:

```
| ?- list_queue([1,2,3,4], Q).

Q = q(s(s(s(s(0)))),[1,2,3,4|_138],_138) ?

yes
| ?-
```

`queue_length(+`*Queue*`, ?`*Length*`)`

Length is the number of elements in *Queue*. Example:

```
| ?- queue_length(q(s(s(s(s(s(0))))),[a,b,c,d,e|R],R), L).

L = 5,
R = _155 ?

yes
```

20 Random Number Generator

This package provides a random number generator. To load the package, enter the query

```
| ?- use_module(library(random)).
```

`random(`-*Number*`)`
: Binds *Number* to a random float in the interval [0.0, 1.0). Note that 1.0 will never be generated.

`random(+`*Lower*`, +`*Upper*`, -`*Number*`)`
: Binds *Number* to a random integer in the interval [*Lower*,*Upper*) if *Lower* and *Upper* are integers. Otherwise *Number* is bound to a random float between *Lower* and *Upper*. *Upper* will never be generated.

`randseq(+`*K*`, +`*N*`, -`*RandomSeq*`)`
: Generates a unordered set of *K* unique integers, chosen randomly in the range *1..N*. *RandomSeq* is not returned in any particular order.

`randset(+`*K*`, +`*N*`, -`*RandomSet*`)`
: Generates an ordered set of *K* unique integers, chosen randomly in the range *1..N*. The set is returned in standard order.

`getrand(`?*State*`)`
: Tries to unify *State* with the term `rand(`*X*,*Y*,*Z*`)` where *X*, *Y*, and *Z* are integers describing the state of the random generator.

`setrand(rand(+`*X*`,+`*Y*`,+`*Z*`))`
: Sets the state of the random generator. *X*, *Y*, and *Z* must be integers in the ranges `[1,30269)`, `[1,30307)`, and `[1,30323)`, respectively.

21 Operating System Utilities

This package contains utilities for invoking services from the operating system. To load the package, enter the query

```
| ?- use_module(library(system)).
```

Certain predicates described below take names of files or directories as arguments. These must be given as atoms, and the predicates below will not call `absolute_file_name/2` on them.

Some predicates are described as invoking the default shell. Specifically this means invoking '`/bin/sh`' on UNIX platforms. On MSDOS, Windows and OS/2, the command interpreter given by the environment variable `COMSPEC` is invoked.

`datime(-`*Datime*`)`
: *Datime* is a timestamp of the form `datime(`*Year*`,`*Month*`,`*Day*`,`*Hour*`,`*Min*`,`*Sec*`)` containing the current date and time. All fields are integers.

`delete_file(+`*FileName*`,+`*Options*`)`
: *FileName* is the name of an existing file or directory. *Options* is a list of options. Possible options are `directory`, `recursive` or `ignore`. If *FileName* is not a directory it is deleted, otherwise if the option `directory` is specified but not `recursive`, the directory will be deleted if it is empty. If `recursive` is specified and *FileName* is a directory, the directory and all its subdirectories and files will be deleted. If the operation fails, an exception is raised unless the `ignore` option is specified.

`delete_file(+`*FileName*`)`
: Equivalent to `delete_file(`*FileName*`,[recursive])`.

`directory_files(+`*Directory*`,-`*FileList*`)`
: *FileList* is the list of entries (files, directories, etc.) in *Directory*.

`make_directory(+`*DirectoryName*`)`
: Makes a new directory.

`environ(?`*Var*`, ?`*Value*`)`
: *Var* is the name of an environment variable, and *Value* is its value. Both are atoms. Can be used to enumerate all current environment variables.

`exec(+`*Command*`, [+`*Stdin*`,+`*Stdout*`,+`*Stderr*`], -`*Pid*`)`
: Passes *Command* to a new default shell process for execution. The standard I/O streams of the new process are connected according to what is specified by the terms `+`*Stdin*, `+`*Stdout*, and `+`*Stderr* respectively. Possible values are:

  `null` — Connected to '`/dev/null`' or equivalent.

  `std` — The standard stream is shared with the calling process. Note that the standard stream may not be referring to a console if the calling process is "windowed". To portably print the output from the subprocess on the Prolog console, `pipe/1` must be used and the program must explicitly read the pipe and write to the console. Similarly for the input to the subprocess.

`pipe(-`*Stream*`)`
: A pipe is created which connects the Prolog stream *Stream* to the standard stream of the new process. It must be closed using `close/1`; it is not closed automatically when the process dies.

Pid is the process identifier of the new process.

On UNIX, the subprocess will be detached provided none of its standard streams is specified as `std`. This means it will not receive an interruption signal as a result of `^C` being typed.

`file_exists(+`*FileName*`)`
: *FileName* is the name of an existing file or directory.

`file_exists(+`*FileName*`, +`*Permissions*`)`
: *FileName* is the name of an existing file or directory which can be accessed according to *Permissions*. *Permissions* is an atom, an integer (see access(2)), or a list of atoms and/or integers. The atoms must be drawn from the list `[read,write,search,exists]`.

`file_property(+`*FileName*`, ?`*Property*`)`
: *FileName* has the property *Property*. The possible properties are:

  `type(`*Type*`)`
  : *Type* is one of `regular`, `directory`, `fifo`, `symlink`, `socket` or `unknown`.

  `size(`*Size*`)`
  : *Size* is the size of *FileName*.

  `mod_time(`*ModTime*`)`
  : *ModTime* is the time of the last modification of *FileName*.

  If *Property* is uninstantiated, the predicate will enumerate the properties on backtracking.

`host_id(-`*HID*`)`
: *HID* is the unique identifier, represented by an atom, of the host executing the current SICStus Prolog process.

`host_name(-`*HostName*`)`
: *HostName* is the standard host name of the host executing the current SICStus Prolog process.

`pid(-`*PID*`)`
: *PID* is the identifier of the current SICStus Prolog process.

`kill(+`*Pid*`, +`*Signal*`)`
: Sends the signal *Signal* to process *Pid*.

`mktemp(+`*Template*`, -`*FileName*`)`
: Interface to the UNIX function mktemp(3). A unique file name is created and unified with *FileName*. *Template* should contain a file name with six trailing `X`s. The unique file name is that template with the six `X`s replaced by a character string.

`popen(+`*Command*`, +`*Mode*`, `*?Stream*`)`
: Interface to the UNIX function `popen(3)`. Passes *Command* to a new default shell process for execution. *Mode* may be either `read` or `write`. In the former case the output from the process is piped to *Stream*. In the latter case the input to the process is piped from *Stream*. *Stream* may be read/written using the ordinary *StreamIO* predicates. It must be closed using `close/1`; it is not closed automatically when the process dies.

`rename_file(+`*OldName*`, +`*NewName*`)`
: *OldName* is the name of an existing file or directory, which will be renamed to *NewName*. If the operation fails, an exception is raised.

`shell`
: Starts a new interactive shell named by the environment variable `SHELL`. The control is returned to Prolog upon termination of the shell process.

`shell(+`*Command*`)`
: Passes *Command* to a new shell named by the environment variable `SHELL` for execution. Succeeds if the return status value is 0.

  On MSDOS, Windows or OS/2, if `SHELL` is defined it is expected to name a UNIX like shell which will be invoked with the argument `-c` *Command*. If `SHELL` is undefined, the shell named by `COMSPEC` will be invoked with the argument `/C` *Command*.

`shell(+`*Command*`, -`*Status*`)`
: Passes *Command* to a new shell named by the environment variable `SHELL` for execution. The status value is returned in *Status*. See also `shell/1` above.

`sleep(+`*Seconds*`)`
: Puts the SICStus Prolog process asleep for *Second* seconds.

`system`
: Starts a new interactive default shell process. The control is returned to Prolog upon termination of the shell process.

`system(+`*Command*`)`
: Passes Command to a new default shell process for execution. Succeeds if the return status value is 0.

`system(+`*Command*`, -`*Status*`)`
: Passes *Command* to a new default shell process for execution. The status value is returned in *Status*.

`tmpnam(-`*FileName*`)`
: Interface to the ANSI C function tmpnam(3). A unique file name is created and unified with *FileName*.

`wait(+`*Pid*`, -`*Status*`)`
: Waits for the child process *Pid* to terminate. The exit status is returned in *Status*. The function is similar to that of the UNIX function `waitpid(3)`.

`working_directory(`*?OldDirectory*`, `*?NewDirectory*`)`
: *OldDirectory* is the current working directory, and the working directory is set to *NewDirectory*. In particular, the goal `working_directory(`*Dir*`,`*Dir*`)` unifies *Dir* with the current working directory without changing anything.

22 Updatable Binary Trees

This package uses binary trees to represent arrays of N elements where N is fixed, unlike `library(arrays)`. To load the package, enter the query

```
| ?- use_module(library(trees)).
```

Binary trees have the following representation: `t` denotes the empty tree, and `t(`*Label*`,`*Left*`,`*Right*`)` denotes the binary tree with label *Label* and children *Left* and *Right*.

`gen_label(`*?Index*`, +`*Tree*`, ?`*Label*`)`
: *Label* labels the *Index*-th element in the *Tree*. Can be used to enumerate all *Labels* by ascending *Index*. Use `get_label/3` instead if *Index* is instantiated.

`get_label(+`*Index*`, +`*Tree*`, ?`*Label*`)`
: *Label* labels the *Index*-th element in the *Tree*.

`list_to_tree(+`*List*`, -`*Tree*`)`
: Constructs a binary *Tree* from *List* where `get_label(`*K*`,`*Tree*`,`*Lab*`)` iff *Lab* is the *K*th element of *List*.

`map_tree(:`*Pred*`, ?`*OldTree*`, ?`*NewTree*`)`
: *OldTree* and *NewTree* are binary trees of the same shape and *Pred(Old,New)* is true for corresponding elements of the two trees.

`put_label(+`*I*`, +`*OldTree*`, +`*Label*`, -`*NewTree*`)`
: Constructs *NewTree* which has the same shape and elements as *OldTree*, except that the *I*-th element is *Label*.

`put_label(+`*I*`, +`*OldTree*`, +`*Label*`, -`*NewTree*`, +`*Label*`)`
: Constructs *NewTree* which has the same shape and elements as *OldTree*, except that the *I*-th element is changed from *OldLabel* to *NewLabel*.

`tree_size(+`*Tree*`, ?`*Size*`)`
: Calculates as *Size* the number of elements in the *Tree*.

`tree_to_list(+`*Tree*`, ?`*List*`)`
: Is the converse operation to `list_to_tree/2`. Any mapping or checking operation can be done by converting the tree to a list, mapping or checking the list, and converting the result, if any, back to a tree.

23 Unweighted Graph Operations

Directed and undirected graphs are fundamental data structures representing arbitrary relationships between data objects. This package provides a Prolog implementation of directed graphs, undirected graphs being a special case of directed graphs.

An unweighted directed graph (ugraph) is represented as a list of (vertex-neighbors) pairs, where the pairs are in standard order (as produced by `keysort` with unique keys) and the neighbors of each vertex are also in standard order (as produced by `sort`), every neighbor appears as a vertex even if it has no neighbors itself, and no vertex is a neighbor to itself.

An undirected graph is represented as a directed graph where for each edge *(U,V)* there is a symmetric edge *(V,U)*.

An edge *(U,V)* is represented as the term *U-V*. *U* and *V* must be distinct.

A vertex can be any term. Two vertices are distinct iff they are not identical (`==`).

A path from *u* to *v* is represented as a list of vertices, beginning with *u* and ending with *v*. A vertex cannot appear twice in a path. A path is maximal in a graph if it cannot be extended.

A tree is a tree-shaped directed graph (all vertices have a single predecessor, except the root node, which has none).

A strongly connected component of a graph is a maximal set of vertices where each vertex has a path in the graph to every other vertex.

Sets are represented as ordered lists (see Chapter 18 [Ordsets], page 207).

To load the package, enter the query

```
| ?- use_module(library(ugraphs)).
```

The following predicates are defined for directed graphs.

`vertices_edges_to_ugraph(+`*Vertices*`, +`*Edges*`, -`*Graph*`)`
: Is true if *Vertices* is a list of vertices, *Edges* is a list of edges, and *Graph* is a graph built from *Vertices* and *Edges*. *Vertices* and *Edges* may be in any order. The vertices mentioned in *Edges* do not have to occur explicitly in *Vertices*. *Vertices* may be used to specify vertices that are not connected by any edges.

`vertices(+`*Graph*`, -`*Vertices*`)`
: Unifies *Vertices* with the vertices in *Graph*.

`edges(+`*Graph*`, -`*Edges*`)`
: Unifies *Edges* with the edges in *Graph*.

`add_vertices(+`*Graph1*`, +`*Vertices*`, -`*Graph2*`)`
: *Graph2* is *Graph1* with *Vertices* added to it.

`del_vertices(+`*Graph1*`, +`*Vertices*`, -`*Graph2*`)`
: *Graph2* is *Graph1* with *Vertices* and all edges to and from them removed from it.

`add_edges(+`*Graph1*`, +`*Edges*`, -`*Graph2*`)`
: *Graph2* is *Graph1* with *Edges* and their "to" and "from" vertices added to it.

`del_edges(+`*Graph1*`, +`*Edges*`, -`*Graph2*`)`
: *Graph2* is *Graph1* with *Edges* removed from it.

`transpose(+`*Graph*`, -`*Transpose*`)`
: *Transpose* is the graph computed by replacing each edge *(u,v)* in *Graph* by its symmetric edge *(v,u)*. Takes O(N^2) time.

`neighbors(+`*Vertex*`, +`*Graph*`, -`*Neighbors*`)`
`neighbours(+`*Vertex*`, +`*Graph*`, -`*Neighbors*`)`
: *Vertex* is a vertex in *Graph* and *Neighbors* are its neighbors.

`complement(+`*Graph*`, -`*Complement*`)`
: *Complement* is the complement graph of *Graph*, i.e. the graph that has the same vertices as *Graph* but only the edges that are not in *Graph*.

`compose(+`*G1*`, +`*G2*`, -`*Composition*`)`
: Computes *Composition* as the composition of two graphs, which need not have the same set of vertices.

`transitive_closure(+`*Graph*`, -`*Closure*`)`
: Computes *Closure* as the transitive closure of *Graph* in O(N^3) time.

`symmetric_closure(+`*Graph*`, -`*Closure*`)`
: Computes *Closure* as the symmetric closure of *Graph*, i.e. for each edge *(u,v)* in *Graph*, add its symmetric edge *(v,u)*. Takes O(N^2) time. This is useful for making a directed graph undirected.

`top_sort(+`*Graph*`, -`*Sorted*`)`
: Finds a topological ordering of a *Graph* and returns the ordering as a list of *Sorted* vertices. Fails iff no ordering exists, i.e. iff the graph contains cycles. Takes O(N^2) time.

`max_path(+`*V1*`, +`*V2*`, +`*Graph*`, -`*Path*`, -`*Cost*`)`
: *Path* is a longest path of cost *Cost* from *V1* to *V2* in *Graph*, there being no cyclic paths from *V1* to *V2*. Takes O(N^2) time.

`min_path(+`*V1*`, +`*V2*`, +`*Graph*`, -`*Path*`, -`*Cost*`)`
: *Path* is a shortest path of cost *Cost* from *V1* to *V2* in *Graph*. Takes O(N^2) time.

`min_paths(+`*Vertex*`, +`*Graph*`, -`*Tree*`)`
: *Tree* is a tree of all the shortest paths from *Vertex* to every other vertex in *Graph*. This is the single-source shortest paths problem.

`path(+`*Vertex*`, +`*Graph*`, -`*Path*`)`
: Given a *Graph* and a *Vertex* of *Graph*, returns a maximal *Path* rooted at *Vertex*, enumerating more paths on backtracking.

`reduce(+`*Graph*`, -`*Reduced*`)`
: *Reduced* is the reduced graph for *Graph*. The vertices of the reduced graph are the strongly connected components of *Graph*. There is an edge in *Reduced* from *u* to *v* iff there is an edge in *Graph* from one of the vertices in *u* to one of the vertices in *v*.

`reachable(+`*Vertex*`, +`*Graph*`, -`*Reachable*`)`
: Given a *Graph* and a *Vertex* of *Graph*, returns the set of vertices that are *reachable* from that *Vertex*, including *Vertex* itself. Takes O(N^2) time.

`random_ugraph(+`*P*`, +`*N*`, -`*Graph*`)`
: Where *P* is a probability, unifies *Graph* with a random graph of vertices *1..N* where each possible edge is included with probability *P*.

The following predicates are defined for undirected graphs only.

`min_tree(+`*Graph*`, -`*Tree*`, -`*Cost*`)`
: *Tree* is a spanning tree of *Graph* with cost *Cost*, if it exists.

`clique(+`*Graph*`, +`*K*`, -`*Clique*`)`
: *Clique* is a maximal clique (complete subgraph) of *N* vertices of *Graph*, where *N*>=*K*. *N* is not necessarily maximal.

`independent_set(+`*Graph*`, +`*K*`, -`*Set*`)`
: *Set* is a maximal independent (unconnected) set of *N* vertices of *Graph*, where *N*>=*K*. *N* is not necessarily maximal.

`coloring(+`*Graph*`, +`*K*`, -`*Coloring*`)`
`colouring(+`*Graph*`, +`*K*`, -`*Coloring*`)`
: *Coloring* is a mapping from vertices to colors *1..N* of *Graph* such that all edges have distinct end colors, where *N*=<*K*. The mapping is represented as an ordered list of *Vertex*-*Color* pairs. *N* is not necessarily minimal.

24 Weighted Graph Operations

A weighted directed graph (wgraph) is represented as a list of (vertex-edgelist) pairs, where the pairs are in standard order (as produced by `keysort` with unique keys), the edgelist is a list of (neighbor-weight) pair also in standard order (as produced by `keysort` with unique keys), every weight is a nonnegative integer, every neighbor appears as a vertex even if it has no neighbors itself, and no vertex is a neighbor to itself.

An undirected graph is represented as a directed graph where for each edge *(U,V)* there is a symmetric edge *(V,U)*.

An edge *(U,V)* with weight *W* is represented as the term *U-(V-W)*. *U* and *V* must be distinct.

A vertex can be any term. Two vertices are distinct iff they are not identical (`==`).

A path from *u* to *v* is represented as a list of vertices, beginning with *u* and ending with *v*. A vertex cannot appear twice in a path. A path is maximal in a graph if it cannot be extended.

A tree is a tree-shaped directed graph (all vertices have a single predecessor, except the root node, which has none).

A strongly connected component of a graph is a maximal set of vertices where each vertex has a path in the graph to every other vertex.

Sets are represented as ordered lists (see Chapter 18 [Ordsets], page 207).

To load the package, enter the query

```
| ?- use_module(library(wgraphs)).
```

The following predicates are defined for directed graphs.

`wgraph_to_ugraph(+`*WeightedGraph*`, -`*Graph*`)`
: *Graph* has the same vertices and edges as *WeightedGraph*, except the edges of *Graph* are unweighted.

`ugraph_to_wgraph(+`*Graph*`, -`*WeightedGraph*`)`
: *WeightedGraph* has the same vertices and edges as *Graph*, except the edges of *WeightedGraph* all have weight 1.

`vertices_edges_to_wgraph(+`*Vertices*`, +`*Edges*`, -`*WeightedGraph*`)`
: *Vertices* is a list of vertices, *Edges* is a list of edges, and *WeightedGraph* is a graph built from *Vertices* and *Edges*. *Vertices* and *Edges* may be in any order. The vertices mentioned in *Edges* do not have to occur explicitly in *Vertices*. *Vertices* may be used to specify vertices that are not connected by any edges.

`vertices(+`*WeightedGraph*`, -`*Vertices*`)`
: Unifies *Vertices* with the vertices in *WeightedGraph*.

`edges(+`*WeightedGraph*`, -`*Edges*`)`
: Unifies *Edges* with the edges in *WeightedGraph*.

`add_vertices(+`*WeightedGraph1*`, +`*Vertices*`, -`*WeightedGraph2*`)`
: *WeightedGraph2* is *WeightedGraph1* with *Vertices* added to it.

`del_vertices(+`*WeightedGraph1*`, +`*Vertices*`, -`*WeightedGraph2*`)`
: *WeightedGraph2* is *WeightedGraph1* with *Vertices* and all edges to and from them removed from it.

`add_edges(+`*WeightedGraph1*`, +`*Edges*`, -`*WeightedGraph2*`)`
: *WeightedGraph2* is *WeightedGraph1* with *Edges* and their "to" and "from" vertices added to it.

`del_edges(+`*WeightedGraph1*`, +`*Edges*`, -`*WeightedGraph2*`)`
: *WeightedGraph2* is *WeightedGraph1* with *Edges* removed from it.

`transpose(+`*WeightedGraph*`, -`*Transpose*`)`
: *Transpose* is the graph computed by replacing each edge *(u,v)* in *WeightedGraph* by its symmetric edge *(v,u)*. It can only be used one way around. Takes O(N^2) time.

`neighbors(+`*Vertex*`, +`*WeightedGraph*`, -`*Neighbors*`)`
`neighbours(+`*Vertex*`, +`*WeightedGraph*`, -`*Neighbors*`)`
: *Vertex* is a vertex in *WeightedGraph* and *Neighbors* are its weighted neighbors.

`transitive_closure(+`*WeightedGraph*`, -`*Closure*`)`
: Computes *Closure* as the transitive closure of *WeightedGraph* in O(N^3) time.

`symmetric_closure(+`*WeightedGraph*`, -`*Closure*`)`
: Computes *Closure* as the symmetric closure of *WeightedGraph*, i.e. for each edge *(u,v)* in *WeightedGraph*, add its symmetric edge *(v,u)*. Takes O(N^2) time. This is useful for making a directed graph undirected.

`top_sort(+`*WeightedGraph*`, -`*Sorted*`)`
: Finds a topological ordering of a *WeightedGraph* and returns the ordering as a list of *Sorted* vertices. Fails iff no ordering exists, i.e. iff the graph contains cycles. Takes O(N^2) time.

`max_path(+`*V1*`, +`*V2*`, +`*WeightedGraph*`, -`*Path*`, -`*Cost*`)`
: *Path* is a maximum-cost path of cost *Cost* from *V1* to *V2* in *WeightedGraph*, there being no cyclic paths from *V1* to *V2*. Takes O(N^2) time.

`min_path(+`*V1*`, +`*V2*`, +`*WeightedGraph*`, -`*Path*`, -`*Cost*`)`
: *Path* is a minimum-cost path of cost *Cost* from *V1* to *V2* in *WeightedGraph*. Takes O(N^2) time.

`min_paths(+`*Vertex*`, +`*WeightedGraph*`, -`*Tree*`)`
: *Tree* is a tree of all the minimum-cost paths from *Vertex* to every other vertex in *WeightedGraph*. This is the single-source minimum-cost paths problem.

`path(+`*Vertex*`, +`*WeightedGraph*`, -`*Path*`)`
: Given a *WeightedGraph* and a *Vertex* of *WeightedGraph*, returns a maximal *Path* rooted at *Vertex*, enumerating more paths on backtracking.

`reduce(+`*WeightedGraph*`, -`*Reduced*`)`
: *Reduced* is the reduced graph for *WeightedGraph*. The vertices of the reduced graph are the strongly connected components of *WeightedGraph*. There is an edge in *Reduced* from *u* to *v* iff there is an edge in *WeightedGraph* from one of the vertices in *u* to one of the vertices in *v*.

`reachable(+`*Vertex*`, +`*WeightedGraph*`, -`*Reachable*`)`
: Given a *WeightedGraph* and a *Vertex* of *WeightedGraph*, returns the set of vertices that are *reachable* from that *Vertex*. Takes O(N^2) time.

`random_wgraph(+`*P*`, +`*N*`, +`*W*`, -`*WeightedGraph*`)`
: Where *P* is a probability, unifies *WeightedGraph* with a random graph of vertices *1..N* where each possible edge is included with probability *P* and random weight in *1..W*.

The following predicate is defined for undirected graphs only.

`min_tree(+`*WeightedGraph*`, -`*Tree*`, -`*Cost*`)`
: *Tree* is a minimum-cost spanning tree of *WeightedGraph* with cost *Cost*, if it exists.

25 Socket I/O

This library package defines a number of predicates manipulating sockets. They are all rather straight-forward interfaces to the corresponding BSD-type socket functions with the same name (except `current_host/1`). The reader should therefore study the appropriate documents for a deeper description.

The *Domain* is either the atom `'AF_INET'` or `'AF_UNIX'`. They correspond directly to the same domains in BSD-type sockets. `'AF_UNIX'` may not be available on non-UNIX platforms.

An *Address* is either `'AF_INET'(Host,Port)` or `'AF_UNIX'(SocketName)`. *Host* is an atomic hostname, *Port* is a portnumber and *SocketName* is an atom denoting a socket. A reader familiar with BSD sockets will understand this immediately.

All streams below can be both read from and written on. All I/O-predicates operating on streams can be used, for example `read/2`, `write/2`, `format/3`, `current_stream/3`, etc. Socket streams are unbuffered on reads and block buffered on writes.

To load the package, enter the query

```
| ?- use_module(library(sockets)).
```

`socket(+`*Domain*`, -`*Socket*`)`
: A socket *Socket* in the domain *Domain* is created.

`socket_close(+`*Socket*`)`
: *Socket* is closed. Sockets used in `socket_connect/2` should not be closed by `socket_close/1` as they will be closed when the corresponding stream is closed.

`socket_bind(+`*Socket*`, 'AF_UNIX'(+`*SocketName*`))`
`socket_bind(+`*Socket*`, 'AF_INET'(?`*Host*`,?`*Port*`))`
: The socket *Socket* is bound to the address. If *Port* is uninstantiated, the operative system picks a port number to which *Port* is bound.

`socket_connect(+`*Socket*`, 'AF_UNIX'(+`*SocketName*`), -`*Stream*`)`
`socket_connect(+`*Socket*`, 'AF_INET'(+`*Host*`,+`*Port*`), -`*Stream*`)`
: The socket *Socket* is connected to the address. *Stream* is a special stream on which items can be both read and written.

`socket_listen(+`*Socket*`, +`*Length*`)`
: The socket *Socket* is defined to have a maximum backlog queue of *Length* pending connections.

`socket_accept(+`*Socket*`, -`*Stream*`)`
`socket_accept(+`*Socket*`, -`*Client*`, -`*Stream*`)`
: The first connection to socket *Socket* is extracted. The stream *Stream* is opened for read and write on this connection. For the `'AF_INET'` domain, *Client* will unified with an atom containing the Internet host address of the connecting entity in numbers-and-dots notation. For other domains, *Client* will not be used.

`socket_select(+`*TermsSockets*`, -`*NewTermsStreams*`, +`*TimeOut*`, +`*Streams*`, -`*ReadStreams*`)`
: The list of streams in *Streams* is checked for readable characters. A stream can be any stream associated with an I/O descriptor. The list *ReadStreams* returns the streams with readable data.

  `socket_select/5` also waits for connections to the sockets specified by *TermsSockets*. This argument should be a list of *Term*`-`*Socket* pairs, where *Term*, which can be any term, is used as an identifier. *NewTermsStreams* is a list of *Term*`-connection(`*Client*`,`*Stream*`)` pairs, where *Stream* is a new stream open for communicating with a process connecting to the socket identified with *Term*, *Client* is the client host address (see `socket_accept/3`).

  If *TimeOut* is instantiated to `off`, the predicate waits until something is available. If *TimeOut* is `S:U` the predicate waits at most `S` seconds and `U` microseconds. Both `S` and `U` must be integers `>=0`. If there is a timeout, *ReadStreams* and *NewTermsStreams* are `[]`.

`socket_select(+`*Sockets*`, -`*NewStreams*`, +`*TimeOut*`, +`*Streams*`, -`*ReadStreams*`)`
`socket_select(+`*Socket*`, -`*NewStream*`, +`*TimeOut*`, +`*Streams*`, -`*ReadStreams*`)`
`socket_select(+`*Sockets*`, -`*NewStreams*`, -`*NewClients*`, +`*TimeOut*`, +`*Streams*`, -`*ReadStreams*`)`
`socket_select(+`*Socket*`, -`*NewStream*`, -`*NewClient*`, +`*TimeOut*`, +`*Streams*`, -`*ReadStreams*`)`
: These forms, which are provided for backward compatibility only, differs in how sockets are specified and new streams returned.

  `socket_select/(5,6)` also wait for connections to the sockets in the list *Sockets*. *NewStreams* is the list of new streams opened for communicating with the connecting processes. *NewClients* is the corresponding list of client host addresses (see `socket_accept/3`).

  The second form requires one socket (not a list) for the first argument and returns a stream, *NewStream*, if a connection is made.

`current_host(?`*HostName*`)`
: *HostName* is unified with the fully qualified name of the machine the process is executing on. The call will also succeed if *HostName* is instantiated to the unqualified name of the machine.

`hostname_address(+`*HostName*`, -`*HostAddress*`)`
`hostname_address(-`*HostName*`, +`*HostAddress*`)`
: The Internet host is resolved given either the host name or address. *HostAddress* should be an atom containing the Internet host address in numbers-and-dots notation. The predicate will fail if the host name or address cannot be resolved.

26 Linda—Process Communication

Linda is a concept for process communication.

For an introduction and a deeper description, see [Carreiro & Gelernter 89a] or [Carreiro & Gelernter 89b], respectively.

One process is running as a server and one or more processes are running as clients. The processes are communicating with sockets and supports networks.

The server is in principle a blackboard on which the clients can write (`out/1`), read (`rd/1`) and remove (`in/1`) data. If the data is not present on the blackboard, the predicates suspend the process until they are available.

There are some more predicates besides the basic `out/1`, `rd/1` and `in/1`. The `in_noblock/1` and `rd_noblock/1` does not suspend if the data is not available—they fail instead. A blocking fetch of a conjunction of data can be done with `in/2` or `rd/2`.

Example: A simple producer-consumer. In client 1:

```
producer :-
       produce(X),
       out(p(X)),
       producer.

produce(X) :- .....
```

In client 2:

```
consumer :-
       in(p(A)),
       consume(A),
       consumer.

consume(A) :- .....
```

Example: Synchronization

```
       ...,
       in(ready),  %Waits here until someone does out(ready)
       ...,
```

Example: A critical region

```
       ...,
       in(region_free),  % wait for region to be free
       critical_part,
       out(region_free), % let next one in
       ...,
```

Example: Reading global data

```
            ...,
            rd(data(Data)),
            ...,

    or, without blocking:
            ...,
            rd_noblock(data(Data)) ->
                  do_something(Data)
            ;     write('Data not available!'),nl
            ),
            ...,
```

Example: Waiting for one of several events

```
            ...,
            in([e(1),e(2),...,e(n)], E),
    %  Here is E instantiated to the first tuple that became available
            ...,
```

26.1 Server

The server is the process running the "blackboard process". It is an ordinary SICStus process which can be run on a separate machine if necessary.

To load the package, enter the query

```
| ?- use_module(library('linda/server')).
```

and start the server with `linda/0` or `linda/1`.

`linda` Starts a Linda-server in this SICStus. The network address is written to current output stream as *Host:PortNumber*.

`linda(+`*Hook*`)`
Starts a Linda-server in this SICStus. When it is started, a goal passed in *Hook* is evaluated. *Hook* must have the form *Address-Goal* where *Address* must be unifiable with *Host:Port* and *Goal* must be instantiated to a goal. **Example**:

```
| ?- linda((Host:Port)-(my_module:mypred(Host,Port))).
```

will call `mypred/2` in module `my_module` when the server is started. `mypred/2` could start the client-processes, save the address for the clients etc. Note that the module must be present in *Goal*.

26.2 Client

The clients are one or more sicstus processes which have connection(s) to the server.

To load the package, enter the query

```
| ?- use_module(library('linda/client')).
```

Some of the following predicates fail if they don't receive an answer from the Linda-server in a reasonable amount of time. That time is set with the predicate `linda_timeout/2`.

`linda_client(+`*Address*`)`
: Establishes a connection to a Linda-server specified by *Address*. The *Address* is of the format *Host:PortNumber* as given by `linda/0` and `linda/1`.

  It is not possible to be connected to two Linda-servers in the same time.

  This predicate can fail due to a timeout.

`close_client`
: Closes the connection to the Linda-server.

`linda_timeout(`*?OldTime*`, `*?NewTime*`)`
: This predicate controls Linda's timeout. *OldTime* is unified with the old timeout and then timeout is set to *NewTime*. The value is either `off` or of the form *Seconds:Milliseconds*. The former value indicates that the timeout mechanism is disabled, that is, eternal waiting. The latter form is the *timeout-time*.

`out(+`*Tuple*`)`
: Places the tuple *Tuple* in Linda's tuple-space.

`in(`*?Tuple*`)`
: Removes the tuple *Tuple* from Linda's tuple-space if it is there. If not, the predicate blocks until it is available (that is, someone performs an `out/1`).

`in_noblock(`*?Tuple*`)`
: Removes the tuple *Tuple* from Linda's tuple-space if it is there. If not, the predicate fails.

  This predicate can fail due to a timeout.

`in(+`*TupleList*`, `*?Tuple*`)`
: As `in/1` but succeeds when either of the tuples in *TupleList* is available. *Tuple* is unified with the fetched tuple. If that unification fails, the tuple is *not* reinserted in the tuple-space.

`rd(`*?Tuple*`)`
: Succeeds if *Tuple* is available in the tuple-space, suspends otherwise until it is available. Compare this with `in/1`: the tuple is *not* removed.

`rd_noblock(`*?Tuple*`)`
: Succeeds if *Tuple* is available in the tuple-space, fails otherwise.

  This predicate can fail due to a timeout.

`rd(+`*TupleList*`, ?`*Tuple*`)`

As `in/2` but does not remove any tuples.

`bagof_rd_noblock(?`*Template*`, +`*Tuple*`, ?`*Bag*`)`

Bag is the list of all instances of *Template* such that *Tuple* exists in the tuple-space.

The behavior of variables in *Tuple* and *Template* is as in `bagof/3`. The variables could be existentially quantified with `^/2` as in `bagof/3`.

The operation is performed as an atomic operation.

This predicate can fail due to a timeout.

Example: Assume that only one client is connected to the server and that the tuple-space initially is empty.

```
| ?- out(x(a,3)), out(x(a,4)), out(x(b,3)), out(x(c,3)).

yes
| ?- bagof_rd_noblock(C-N, x(C,N), L).

C = _32,
L = [a-3,a-4,b-3,c-3],
N = _52 ?

yes
| ?- bagof_rd_noblock(C, N^x(C,N), L).

C = _32,
L = [a,a,b,c],
N = _48 ?

yes
```

27 External Storage of Terms (External Database)

This library handles storage and retrieval of terms on files. By using indexing, the store/retrieve operations are efficient also for large data sets.

The package is loaded by the query

```
| ?- use_module(library(db)).
```

27.1 Basics

The idea is to get a behavior similar to `assert/1`, `retract/1` and `clause/2` but the terms are stored on files instead of in primary memory.

The differences compared with the internal database are:

- A *database* must be opened before any access and closed after the last access. (There are special predicates for this: `db_open` and `db_close`.)
- The indexing is specified when the database is created. It is possible to index on other parts of the term than just the functor and first argument.
- Changes affect the database immediately.
- The database will store variables with frozen goals as ordinary variables.

Some commercial databases can't store non-ground terms or more than one instance of a term. The SICStus database can however store terms of either kind.

The database is kept in a secure state and the last stored term is safe even if the SICStus process dies (machine rebooted, process killed, `halt/0`, power failure...) (but see Section 27.2 [Current Limitations], page 235).

27.2 Current Limitations

- Only one process can open the same database.
- The terms are not necessarily fetched in the same order as they were stored.
- If the process dies *during* the `db_store` the database *could* be inconsistent.
- At the user's option, a database can be operated in an insecure state where modified pages are not immediately flushed to disk. This is useful e.g. when a database is created by storing a large number of terms. See `db_buffering/(2-3)`.

27.3 The DB-Spec—Informal Description

The *db-spec* defines which parts of a term that is used for indexing in a database. It is a structure with the functor `on` or `off`. The arguments are `on`, `off` or the same kind of structure.

The db-spec is compared with the *indexed term* and every argument where there is an `on` in the db-spec is indexed.

If the db-spec is of lower arity than the indexed term, the last part of the indexed term is skipped or vice versa.

The idea of a db-spec are illustrated with a few examples. (A section further down explains the db-spec in a more formal way).

DB-Spec *Term (the parts with indexing are underlined)*

```
on(on)           a(b)   a(b,c)  a   [a,b(c),d]    /* as Prolog */
                 - -    - -     -   --

on(on,on)        a(b)   a(b,c)  a(b,c,d)  a(b,c(d))   [a,b(c),d]
                 - -    - - -   - - -     - - -       ---

on(off,on(on))   a(b)   a(b,c)  a(b,c,d)  a(b,c(d)) [a,b(c),d]
                 -      -   -   -   -     -   - -   - --
```

27.4 Predicates

The following conventions are used in the predicate descriptions below.

- *Mode* is either `update` or `read`
- *DBRef* is a reference to an open database. The reference is returned when the database is opened. The reference becomes invalid after the database has been closed.
- *TermRef* is a reference to a term in a given database. The reference is returned when a term is stored. The reference stays valid even after the database is closed and hence can be stored permanently as part of another term.

  A term generally corresponds to several alias references. `db_canonical/(2,3)` can be used to check whether two *TermRef*s refer to the same term.
- *Spec* is a description of the indexing, see the description of db-specs below.
- *Term* is any Prolog term.

`db_open(+`*Name*`, +`*Mode*`, -`*DBref*`)`
`db_open(+`*Name*`, +`*Mode*`, ?`*Spec*`, -`*DBref*`)`

Opens a database with the name *Name*. The database physically consists of a subdirectory with the same name, containing the files that make up the database. If the subdirectory does not exist, it is created. In that case *Mode* must be `update`.

`db_open/4`:

The db-spec *Spec* must be ground if opening a new database. If an existing database is opened, *Spec* is unified with the db-spec given when the database was created. If the unification fails the predicate fails and an error is raised.

`db_open/3`:

On creating a new database, the db-spec is `on(on)`, the same kind of indexing as in the internal database. When opening an existing database, any db-spec from the database is accepted.

`db_close`
`db_close(+`*DBref*`)`

Closes a database. `db_close/0` closes the *default database*. Note that after `db_close/0` there is no default database. `abort/0` does not close databases.

`set_default_db(+`*DBref*`)`
`set_default_db(+`*Name*`)`

Sets the database *Name* or *DBref* to be the default database regardless of whether there already is one or not.

`get_default_db(?`*DBref*`)`

Unifies *DBref* with the default database.

`current_db(?`*Name*`, ?`*Mode*`, ?`*Spec*`, ?`*DBref*`)`

Unifies the arguments with the open databases. This predicate can be used for enumerating all currently open databases through backtracking.

`db_store(+`*Term*`, -`*TermRef*`)`
`db_store(+`*DBref*`, +`*Term*`, -`*TermRef*`)`

Stores *Term* in the database *DBref*, which defaults to the default database. *TermRef* is unified with a corresponding term reference.

`db_fetch(?`*Term*`, ?`*TermRef*`)`
`db_fetch(+`*DBref*`, ?`*Term*`, ?`*TermRef*`)`

Unifies *Term* with a term from the database *DBref*, which defaults to the default database. At the same time, *TermRef* is unified with a corresponding term reference. Backtracking over the predicate unifies with all terms matching *Term*. If you simply want to find all matching terms, it is more efficient to use `db_findall/(2-3)`.

If *TermRef* and *DBref* are instantiated (and the referenced term is not erased), the referenced term is read and unified with *Term*.

`db_findall(?`*Term*`, ?`*TermList*`)`
`db_findall(+`*DBref*`, ?`*Term*`, ?`*TermList*`)`

Unifies *TermList* with the list of all terms matching *Term* from the database *DBref*, which defaults to the default database. The list is guaranteed to be free of duplicates, except in the cases where the same term has actually been stored more than once.

`db_erase(+`*TermRef*`)`
`db_erase(+`*DBref*`, +`*TermRef*`)`

Deletes the term in the database *DBref*, which defaults to the default database, that is referenced by *TermRef* if it is not already deleted.

`db_canonical(+`*TermRef*`, -`*TermID*`)`
`db_canonical(+`*DBref*`, +`*TermRef*`, -`*TermID*`)`
: Returns the canonical term identifier for the term in *DBref*, which defaults to the default database, that is referenced by *TermRef*.

`db_buffering(`*?Old*`, `*?New*`)`
`db_buffering(+`*DBref*`, `*?Old*`, `*?New*`)`
: Unifies *Old* with the current buffering mode of *DBref*, which defaults to the defaults database, and sets its buffering mode to *New*, which must be `on` or `off`. Buffering is initially `off`. When it is `on`, modified pages are not immediately flushed to disk, enabling faster execution but with a higher risk of inconsistencies if a crash occurs. `db_close` always flushes any modified pages.

27.5 An Example Session

```
| ?- db_open(my_db,update,on(on),R), set_default_db(R).

R = '$db'(1411928) ?

yes
| ?- db_store(a(b),_).

yes
| ?- db_store(a(c),_).

yes
| ?- db_fetch(X,_).

X = a(b) ? ;

X = a(c) ? ;

no
| ?- current_db(A,B,C,D).

A = my_db,
B = update,
C = on(on),
D = '$db'(1411928) ? ;

no
| ?- db_close.

yes
```

27.6 The Db-Spec

A db-spec is on of the following terms:

- the atom `on`, or
- the atom `off`, or
- a structure with functor `on` or `off` and every argument being a db-spec.

The following table defines the way indices are calculated a bit more formally. The table defines the index as a function *INDEX* of the db-spec and the indexed term.

The column *Index Calculation* describes the procedure: a "yes" means that some primitive function is used, such as a hash function; "*INDEX(s,t)*" means simply that the function definition (the table) is applied again, but with a new db-spec s and a new term t; I means `on` or `off`.

DB-Spec	*Indexed Term*	*Index Calculation*
`off`	`any`	`no`
`on`	`any`	`yes (on principal functor)`
I`(...)`	`atomic`	`yes`
I`(...)`	`variable`	`yes`
I`(`*S1*`,`*S2*`...`*Sn*`)`	*F*`(`*A1*`,`*A2*`...`*Am*`)`	`yes (on principal functor)`
		INDEX(Si,Ai) `for 1 =<` *i* `=<` `min(`*n*`,`*m*`)`

Every term is stored together with a set of "keywords" for indexing purposes. The space overhead is approximately 16 bytes per keyword per term. The number of keywords stored depends on the db-spec and on the term being indexed. The following table defines the number of keywords as a function K of the db-spec and the indexed term.

DB-Spec	*Indexed Term*	*Number of keywords*
`off`	`any`	`0`
`on`	`any`	`2`
I`(...)`	`atomic`	`2`
I`(...)`	`variable`	`2`
I`(`*S1*`,...,`*Sn*`)`	*F*`(`*A1*`,...,`*Am*`)`	`1+`*K(S1,A1)*...*K(Sj,Aj)*`, j=min(`*n*`,`*m*`)`

28 Boolean Constraint Solver

The clp(B) system provided by this library module is an instance of the general Constraint Logic Programming scheme introduced in [Jaffar & Michaylov 87]. It is a solver for constraints over the Boolean domain, i.e. the values 0 and 1. This domain is particularly useful for modeling digital circuits, and the constraint solver can be used for verification, design, optimization etc. of such circuits.

To load the solver, enter the query:

```
| ?- use_module(library(clpb)).
```

The solver contains predicates for checking the consistency and entailment of a constraint wrt. previous constraints, and for computing particular solutions to the set of previous constraints.

The underlying representation of Boolean functions is based on Boolean Decision Diagrams [Bryant 86]. This representation is very efficient, and allows many combinatorial problems to be solved with good performance.

Boolean expressions are composed from the following operands: the constants 0 and 1 (`FALSE` and *TRUE*), logical variables, and symbolic constants, and from the following connectives. *P* and *Q* are Boolean expressions, *X* is a logical variable, *Is* is a list of integers or integer ranges, and *Es* is a list of Boolean expressions:

`~` *P*
: True if *P* is false.

P `*` *Q*
: True if *P* and *Q* are both true.

P `+` *Q*
: True if at least one of *P* and *Q* is true.

P `#` *Q*
: True if exactly one of *P* and *Q* is true.

X `^` *P*
: True if there exists an *X* such that *P* is true. Same as *P*`[`*X*`/0] + `*P*`[`*X*`/1]`.

P `=:=` *Q*
: Same as `~`*P* `#` *Q*.

P `=\=` *Q*
: Same as *P* `#` *Q*.

P `=<` *Q*
: Same as `~`*P* `+` *Q*.

P `>=` *Q*
: Same as *P* `+ ~`*Q*.

P `<` *Q*
: Same as `~`*P* `*` *Q*.

P `>` *Q*
: Same as *P* `* ~`*Q*.

`card(`*Is*`,` *Es*`)`
: True if the number of true expressions in *Es* is a member of the set denoted by *Is*.

Symbolic constants (Prolog atoms) denote parametric values and can be viewed as all-quantified variables whose quantifiers are placed outside the entire expression. They are useful for forcing certain variables of an equation to be treated as input parameters.

28.1 Solver Interface

The following predicates are defined:

`sat(+`*Expression*`)`
: *Expression* is a Boolean expression. This checks the consistency of the expression wrt. the accumulated constraints, and, if the check succeeds, *tells* the constraint that the expression be true.

  If a variable X, occurring in the expression, is subsequently unified with some term T, this is treated as a shorthand for the constraint

```
?- sat(X=:=T).
```

`taut(+`*Expression*`, ?`*Truth*`)`
: *Expression* is a Boolean expression. This *asks* whether the expression is now entailed by the accumulated constraints (*Truth*=1), or whether its negation is entailed by the accumulated constraints (*Truth*=0). Otherwise, it fails.

`labeling(+`*Variables*`)`
: *Variables* is a list of variables. The variables are instantiated to a list of 0s and 1s, in a way that satisfies any accumulated constraints. Enumerates all solutions by backtracking, but creates choicepoints only if necessary.

28.2 Examples

28.2.1 Example 1

```
| ?- sat(X + Y).

sat(X=\=_A*Y#Y) ?
```

illustrates three facts. First, any accumulated constraints affecting the top-level variables are displayed as floundered goals, since the query is not true for all `X` and `Y`. Secondly, accumulated constraints are displayed as `sat(`V`=:=`*Expr*`)` or `sat(`V`=\=`*Expr*`)` where V is a variable and *Expr* is a "polynomial", i.e. an exclusive or of conjunctions of variables and constants. Thirdly, `_A` had to be introduced as an artificial variable, since `Y` cannot be expressed as a function of `X`. That is, `X + Y` is true iff there exists an `_A` such that `X=\=_A*Y#Y`. Let's check it!

```
| ?- taut(_A ^ (X=\=_A*Y#Y) =:= X + Y, T).

T = 1 ?
```

verifies the above answer. Notice that the formula in this query is a tautology, and so it is entailed by an empty set of constraints.

28.2.2 Example 2

```
| ?- taut(A =< C, T).

no
| ?- sat(A =< B), sat(B =< C), taut(A =< C, T).

T = 1,
sat(A=:=_A*_B*C),
sat(B=:=_B*C) ?

| ?- taut(a, T).

T = 0 ?

yes
| ?- taut(~a, T).

T = 0 ?
```

illustrates the entailment predicate. In the first query, the expression "A implies C" is neither known to be true nor false, so the query fails. In the second query, the system is told that "A implies B" and "B implies C", so "A implies C" is entailed. The expressions in the third and fourth queries are to be read "for each a, a is true" and "for each a, a is false", respectively, and so `T = 0` in both cases since both are unsatisfiable. This illustrates the fact that the implicit universal quantifiers introduced by symbolic constants are placed in front of the entire expression.

28.2.3 Example 3

```
| ?- [user].
| adder(X, Y, Sum, Cin, Cout) :-
     sat(Sum =:= card([1,3],[X,Y,Cin])),
     sat(Cout =:= card([2-3],[X,Y,Cin])).
| {user consulted, 40 msec 576 bytes}

yes
| ?- adder(x, y, Sum, cin, Cout).

sat(Sum=:=cin#x#y),
sat(Cout=:=x*cin#x*y#y*cin) ?

yes
| ?- adder(x, y, Sum, 0, Cout).

sat(Sum=:=x#y),
sat(Cout=:=x*y) ?

yes
```

```
| ?- adder(X, Y, 0, Cin, 1), labeling([X,Y,Cin]).

Cin = 0,
X = 1,
Y = 1 ? ;

Cin = 1,
X = 0,
Y = 1 ? ;

Cin = 1,
X = 1,
Y = 0 ? ;
```

illustrates the use of cardinality constraints and models a one-bit adder circuit. The first query illustrates how representing the input signals by symbolic constants forces the output signals to be displayed as functions of the inputs and not vice versa. The second query computes the simplified functions obtained by setting carry-in to 0. The third query asks for particular input values satisfying sum and carry-out being 0 and 1, respectively.

28.2.4 Example 4

The predicate `fault/3` below describes a 1-bit adder consisting of five gates, with at most one faulty gate. If one of the variables `Fi` is equal to 1, the corresponding gate is faulty, and its output signal is undefined (i.e., the constraint representing the gate is relaxed).

Assuming that we have found some incorrect output from a circuit, we are interesting in finding the faulty gate. Two instances of incorrect output are listed in `fault_ex/2`:

```
fault([F1,F2,F3,F4,F5], [X,Y,Cin], [Sum,Cout]) :-
        sat(
                    card([0-1],[F1,F2,F3,F4,F5]) *
                    (F1 + (U1 =:= X * Cin)) *
                    (F2 + (U2 =:= Y * U3)) *
                    (F3 + (Cout =:= U1 + U2)) *
                    (F4 + (U3 =:= X # Cin)) *
                    (F5 + (Sum =:= Y # U3))
                ).

fault_ex(1, Faults) :- fault(Faults, [1,1,0], [1,0]).
fault_ex(2, Faults) :- fault(Faults, [1,0,1], [0,0]).
```

To find the faulty gates, we run the query

```
| ?- fault_ex(I,L), labeling(L).

I = 1,
L = [0,0,0,1,0] ? ;
```

```
I = 2,
L = [1,0,0,0,0] ? ;

I = 2,
L = [0,0,1,0,0] ? ;

no
```

Thus for input data `[1,1,0]`, gate 4 must be faulty. For input data `[1,0,1]`, either gate 1 or gate 3 must be faulty.

To get a symbolic representation of the outputs interms of the input, we run the query

```
| ?- fault([0,0,0,0,0], [x,y,cin], [Sum,Cout]).

sat(Cout=:=x*cin#x*y#y*cin),
sat(Sum=:=cin#x#y)
```

which shows that the sum and carry out signals indeed compute the intended functions if no gate is faulty.

29 Constraint Logic Programming over Rationals or Reals

29.1 Introduction

The clp(Q,R) system described in this document is an instance of the general Constraint Logic Programming scheme introduced by [Jaffar & Michaylov 87].

The implementation is at least as complete as other existing clp(R) implementations: It solves linear equations over rational or real valued variables, covers the lazy treatment of nonlinear equations, features a decision algorithm for linear inequalities that detects implied equations, removes redundancies, performs projections (quantifier elimination), allows for linear dis-equations, and provides for linear optimization.

The full clp(Q,R) distribution, including a stand-alone manual and an examples directory that is possibly more up to date than the version in the SICStus Prolog distribution, is available from: `http://www.ai.univie.ac.at/clpqr/`.

Referencing this Software

When referring to this implementation of clp(Q,R) in publications, you should use the following reference:

> Holzbaur C.: OFAI clp(q,r) Manual, Edition 1.3.3, Austrian Research Institute for Artificial Intelligence, Vienna, TR-95-09, 1995.

Acknowledgments

The development of this software was supported by the Austrian *Fonds zur Foerderung der Wissenschaftlichen Forschung* under grant P9426-PHY. Financial support for the Austrian Research Institute for Artificial Intelligence is provided by the Austrian Federal Ministry for Science and Research.

We include a collection of examples that has been distributed with the Monash University version of clp(R) [Heintze et al. 87], and its inclusion into this distribution was kindly permitted by Roland Yap.

29.2 Solver Interface

Until rational numbers become first class citizens in SICStus Prolog, rational arithmetics has to be emulated. Because of the emulation it is too expensive to support arithmetics with automatic coercion between all sorts of numbers, like you find it in CommonLisp, for example.

You must choose whether you want to operate in the field of Q (Rationals) or R (Reals):

```
| ?- use_module(library(clpq)).

or

| ?- use_module(library(clpr)).
```

You can also load both modules, but the exported predicates listed below will name-clash (see Section 4.4 [Importation], page 49). You can avoid the interactive resolution dialog if the importation is skipped, e.g. via: `use_module(library(clpq),[]),use_module(library(clpr),[])`.

Notational Conventions

Throughout this chapter, the prompts `clp(q) ?-` and `clp(r) ?-` are used to differentiate between clp(Q) and clp(R) in exemplary interactions.

In general there are many ways to express the same linear relationship. This degree of freedom is manifest in the fact that the printed manual and an actual interaction with the current version of clp(Q,R) may show syntactically different answer constraints, despite the fact the same semantic relationship is being expressed. There are means to control the presentation; see Section 29.5.1 [Variable Ordering], page 260. The approximative nature of floating point numbers may also produce numerical differences between the text in this manual and the actual results of clp(R), for a given edition of the software.

29.2.1 Solver Predicates

The solver interface for both Q and R consists of the following predicates which are exported from `module(linear)`.

`{+`*Constraint*`}`

Constraint is a term accepted by the the grammar below. The corresponding constraint is added to the current constraint store and checked for satisfiability. Use the module prefix to distinguish the solvers if both clp(Q) and clp(R) were loaded

```
| ?- clpr:{Ar+Br=10}, Ar=Br, clpq:{Aq+Bq=10}, Aq=Bq.

Aq = 5,
Ar = 5.0,
Bq = 5,
Br = 5.0
```

Although clp(Q) and clp(R) are independent modules, you are asking for trouble if you (accidently) share variables between them:

```
| ?- clpr:{A+B=10}, clpq:{A=B}.
{TYPE ERROR: _118=5.0 - arg 2: expected 'a rational number', found 5.0}
```

This is because both solvers eventually compute values for the variables and Reals are incompatible with Rationals.

Here is the constraint grammar:

Constraint -->	*C*	
	| *C* , *C*	conjunction
C -->	*Expr* =:= *Expr*	equation
	| *Expr* = *Expr*	equation
	| *Expr* < *Expr*	strict inequation
	| *Expr* > *Expr*	strict inequation
	| *Expr* =< *Expr*	nonstrict inequation
	| *Expr* >= *Expr*	nonstrict inequation
	| *Expr* =\= *Expr*	disequation
Expr -->	*variable*	Prolog variable
	| *number*	floating point or integer
	| + *Expr*	unary plus
	| - *Expr*	unary minus
	| *Expr* + *Expr*	addition
	| *Expr* - *Expr*	subtraction
	| *Expr* * *Expr*	multiplication
	| *Expr* / *Expr*	division
	| *abs(Expr)*	absolute value
	| *sin(Expr)*	trigonometric sine
	| *cos(Expr)*	trigonometric cosine
	| *tan(Expr)*	trigonometric tangent
	| *pow(Expr,Expr)*	raise to the power
	| *exp(Expr,Expr)*	raise to the power
	| *min(Expr,Expr)*	minimum of the two arguments
	| *max(Expr,Expr)*	maximum of the two arguments
	| *#(Const)*	symbolic numerical constants

Conjunctive constraints `{C,C}` have been made part of the syntax to control the granularity of constraint submission, which will be exploited by future versions of this software. Symbolic numerical constants are provided for compatibility only; see Section 29.7.1 [Monash Examples], page 266.

`entailed(+`*Constraint*`)`

Succeeds iff the linear *Constraint* is entailed by the current constraint store. This predicate does not change the state of the constraint store.

```
clp(q) ?- {A =< 4}, entailed(A=\=5).

{A=<4}
yes
```

```
clp(q) ?- {A =< 4}, entailed(A=\=3).

no
```

`inf(+`*Expr*`, -`*Inf*`)`
: Computes the infimum of the linear expression *Expr* and unifies it with *Inf*. Failure indicates unboundedness.

`sup(+`*Expr*`, -`*Sup*`)`
: Computes the supremum of the linear expression *Expr* and unifies it with *Sup*. Failure indicates unboundedness.

```
clp(q) ?- { 2*X+Y =< 16, X+2*Y =< 11,
            X+3*Y =< 15, Z = 30*X+50*Y
          }, sup(Z, Sup).

Sup = 310,
{Z=30*X+50*Y},
{X+1/2*Y=<8},
{X+3*Y=<15},
{X+2*Y=<11}
```

`minimize(+`*Expr*`)`
: Computes the infimum of the linear expression *Expr* and equates it with the expression, i.e. as if defined as:

```
minimize(Expr) :- inf(Expr, Expr).
```

`maximize(+`*Expr*`)`
: Computes the supremum of the linear expression *Expr* and equates it with the expression.

```
clp(q) ?- { 2*X+Y =< 16, X+2*Y =< 11,
            X+3*Y =< 15, Z = 30*X+50*Y
          }, maximize(Z).

X = 7,
Y = 2,
Z = 310
```

`bb_inf(+`*Ints*`, +`*Expr*`, -`*Inf*`)`
: Computes the infimum of the linear expression *Expr* under the additional constraint that all of variables in the list *Ints* assume integral values at the infimum. This allows for the solution of mixed integer linear optimization problems; see Section 29.8 [MIP], page 268.

```
clp(q) ?- {X >= Y+Z, Y > 1, Z > 1}, bb_inf([Y,Z],X,Inf).

Inf = 4,
{Y>1},
{Z>1},
{X-Y-Z>=0}
```

`bb_inf(+`*Ints*`, +`*Expr*`, -`*Inf*`, -`*Vertex*`, +`*Eps*`)`

Computes the infimum of the linear expression *Expr* under the additional constraint that all of variables in the list *Ints* assume integral values at the infimum. *Eps* is a positive number between 0 and 0.5 that specifies how close a number X must be to the next integer to be considered integral: `abs(round(X)-X) < Eps`. The predicate `bb_inf/3` uses `Eps = 0.001`. With clp(Q), `Eps = 0` makes sense. *Vertex* is a list of the same length as *Ints* and contains the (integral) values for *Ints*, such that the infimum is produced when assigned. Note that this will only generate one particular solution, which is different from the situation with `minimize/1`, where the general solution is exhibited.

`ordering(+`*Spec*`)`

Provides a means to control one aspect of the presentation of the answer constraints; see Section 29.5.1 [Variable Ordering], page 260.

`dump(+`*Target*`, -`*NewVars*`, -`*CodedAnswer*`)`

Reflects the constraints on the target variables into a term, where *Target* and *NewVars* are lists of variables of equal length and *CodedAnswer* is the term representation of the projection of constraints onto the target variables where the target variables are replaced by the corresponding variables from *NewVars* (see Section 29.5.2 [Turning Answers into Terms], page 261).

```
clp(q) ?- {A+B =< 10, A>=4},
          dump([A,B],Vs,Cs),
          dump([B],Bp,Cb).

Cb = [_A=<6],
Bp = [_A],
Cs = [_B>=4,_C+_B=<10],
Vs = [_C,_B],
{A>=4},
{A+B=<10}
```

The current version of `dump/3` is incomplete with respect to nonlinear constraints. It only reports nonlinear constraints that are connected to the target variables. The following example has no solution. From the top level's report we have a chance to deduce this fact, but `dump/3` currently has no means to collect global constraints ...

```
q(X) :-
        {X>=10},
        {sin(Z)>3}.

clp(r) ?- q(X), dump([X],V,C).

C = [_A>=10.0],
V = [_A],
clpr:{3.0-sin(_B)<0.0},
{X>=10.0}
```

29.2.2 Unification

Equality constraints are added to the store implicitly each time variables that have been mentioned in explicit constraints are bound - either to another such variable or to a number.

```
clp(r) ?- {2*A+3*B=C/2}, C=10.0, A=B.

A = 1.0,
B = 1.0,
C = 10.0
```

Is equivalent modulo rounding errors to

```
clp(r) ?- {2*A+3*B=C/2, C=10, A=B}.

A = 1.0,
B = 0.9999999999999999,
C = 10.0
```

The shortcut bypassing the use of `{}/1` is allowed and makes sense because the interpretation of this equality in Prolog and clp(R) coincides. In general, equations involving interpreted functors, `+/2` in this case, must be fed to the solver explicitly:

```
clp(r) ?- X=3.0+1.0, X=4.0.

no
```

Further, variables known by clp(R) may be bound directly to floats only. Likewise, variables known by clp(Q) may be bound directly to rational numbers only; see Section 29.9.1.1 [Rationals], page 269. Failing to do so is rewarded with an exception:

```
clp(q) ?- {2*A+3*B=C/2}, C=10.0, A=B.
{TYPE ERROR: _165=10.0 - arg 2: expected 'a rational number', found 10.0}
```

This is because `10.0` is not a rational constant. To make clp(Q) happy you have to say:

```
clp(q) ?- {2*A+3*B=C/2}, C=rat(10,1), A=B.

A = 1,
B = 1,
C = 10
```

If you use `{}/1`, you don't have to worry about such details. Alternatively, you may use the automatic expansion facility, check Section 29.7 [Syntactic Sugar], page 265.

29.2.3 Feedback and Bindings

What was covered so far was how the user populates the constraint store. The other direction of the information flow consists of the success and failure of the above predicates and the binding of variables to numerical values and the aliasing of variables. Example:

```
clp(r) ?- {A-B+C=10, C=5+5}.

B = A,
C = 10.0
```

The linear constraints imply `A=B` and the solver consequently exports this binding to the Prolog world, which is manifest in the fact that the test `A==B` will succeed. More about answer presentation in Section 29.5 [Projection], page 259.

29.3 Linearity and Nonlinear Residues

The clp(Q,R) system is restricted to deal with linear constraints because the decision algorithms for general nonlinear constraints are prohibitively expensive to run. If you need this functionality badly, you should look into symbolic algebra packages. Although the clp(Q,R) system cannot solve nonlinear constraints, it will collect them faithfully in the hope that through the addition of further (linear) constraints they might get simple enough to solve eventually. If an answer contains nonlinear constraints, you have to be aware of the fact that success is qualified modulo the existence of a solution to the system of residual (nonlinear) constraints:

```
clp(r) ?- {sin(X) = cos(X)}.

clpr:{sin(X)-cos(X)=0.0}
```

There are indeed infinitely many solutions to this constraint (`X = 0.785398 + n*Pi`), but clp(Q,R) has no direct means to find and represent them.

The systems goes through some lengths to recognize linear expressions as such. The method is based on a normal form for multivariate polynomials. In addition, some simple isolation axioms, that can be used in equality constraints, have been added. The current major limitation of the method is that full polynomial division has not been implemented. Examples:

This is an example where the isolation axioms are sufficient to determine the value of X.

```
clp(r) ?- {sin(cos(X)) = 1/2}.

X = 1.0197267436954502
```

If we change the equation into an inequation, clp(Q,R) gives up:

```
clp(r) ?- {sin(cos(X)) < 1/2}.

clpr:{sin(cos(X))-0.5<0.0}
```

The following is easy again:

```
clp(r) ?- {sin(X+2+2)/sin(4+X) = Y}.

Y = 1.0
```

And so is this:

```
clp(r) ?- {(X+Y)*(Y+X)/X = Y*Y/X+99}.

{Y=49.5-0.5*X}
```

An ancient symbol manipulation benchmark consists in rising the expression `X+Y+Z+1` to the 15th power:

```
clp(q) ?-  {exp(X+Y+Z+1,15)=0}.
clpq:{Z^15+Z^14*15+Z^13*105+Z^12*455+Z^11*1365+Z^10*3003+...
       ... polynomial continues for a few pages ...
       =0}
```

Computing its roots is another story.

29.3.1 How Nonlinear Residues are made to disappear

Binding variables that appear in nonlinear residues will reduce the complexity of the nonlinear expressions and eventually results in linear expressions:

```
clp(q) ?- {exp(X+Y+1,2) = 3*X*X+Y*Y}.

clpq:{Y*2-X^2*2+Y*X*2+X*2+1=0}
```

Equating X and Y collapses the expression completely and even determines the values of the two variables:

```
clp(q) ?- {exp(X+Y+1,2) = 3*X*X+Y*Y}, X=Y.

X = -1/4,
Y = -1/4
```

29.3.2 Isolation Axioms

These axioms are used to rewrite equations such that the variable to be solved for is moved to the left hand side and the result of the evaluation of the right hand side can be assigned to the variable. This allows, for example, to use the exponentiation operator for the computation of roots and logarithms, see below.

`A = B * C` Residuates unless B or C is ground or A and B or C are ground.

`A = B / C` Residuates unless C is ground or A and B are ground.

`X = min(Y,Z)`
Residuates unless Y and Z are ground.

`X = max(Y,Z)`
Residuates unless Y and Z are ground.

`X = abs(Y)`
Residuates unless Y is ground.

`X = pow(Y,Z)`, `X = exp(Y,Z)`
Residuates unless any pair of two of the three variables is ground. Example:

```
clp(r) ?- { 12=pow(2,X) }.

X = 3.5849625007211565

clp(r) ?- { 12=pow(X,3.585) }.

X = 1.9999854993443926

clp(r) ?- { X=pow(2,3.585) }.

X = 12.000311914286545
```

`X = sin(Y)`
Residuates unless X or Y is ground. Example:

```
clp(r) ?- { 1/2 = sin(X) }.

X = 0.5235987755982989
```

`X = cos(Y)`
Residuates unless X or Y is ground.

`X = tan(Y)`
Residuates unless X or Y is ground.

29.4 Numerical Precision and Rationals

The fact that you can switch between clp(R) and clp(Q) should solve most of your numerical problems regarding precision. Within clp(Q), floating point constants will be coerced into rational numbers automatically. Transcendental functions will be approximated with rationals. The precision of the approximation is limited by the floating point precision. These two provisions allow you to switch between clp(R) and clp(Q) without having to change your programs.

What is to be kept in mind however is the fact that it may take quite big rationals to accommodate the required precision. High levels of precision are for example required if your linear program is ill-conditioned, i.e., in a full rank system the determinant of the coefficient matrix is close to

zero. Another situation that may call for elevated levels of precision is when a linear optimization problem requires exceedingly many pivot steps before the optimum is reached.

If your application approximates irrational numbers, you may be out of space particularly soon. The following program implements N steps of Newton's approximation for the square root function at point 2.

```
%
% from file: library('clpqr/examples/root')
%
root(N, R) :-
  root(N, 1, R).

root(0, S, R) :- !, S=R.
root(N, S, R) :-
  N1 is N-1,
  { S1 = S/2 + 1/S },
  root(N1, S1, R).
```

It is known that this approximation converges quadratically, which means that the number of correct digits in the decimal expansion roughly doubles with each iteration. Therefore the numerator and denominator of the rational approximation have to grow likewise:

```
clp(q) ?- use_module(library('clpqr/examples/root')).
clp(q) ?- root(3,R),print_decimal(R,70).
1.4142156862 7450980392 1568627450 9803921568 6274509803 9215686274
5098039215

R = 577/408

clp(q) ?- root(4,R),print_decimal(R,70).
1.4142135623 7468991062 6295578890 1349101165 5962211574 4044584905
0192000543

R = 665857/470832

clp(q) ?- root(5,R),print_decimal(R,70).
1.4142135623 7309504880 1689623502 5302436149 8192577619 7428498289
4986231958

R = 886731088897/627013566048

clp(q) ?- root(6,R),print_decimal(R,70).
1.4142135623 7309504880 1688724209 6980785696 7187537723 4001561013
1331132652

R = 1572584048032918633353217/1111984844349868137938112

clp(q) ?- root(7,R),print_decimal(R,70).
```

```
1.4142135623 7309504880 1688724209 6980785696 7187537694 8073176679
7379907324

R = 4946041176255201878775086487573351061418968498177 /
    3497379255757941172020851852070562919437964212608
```

Iterating for 8 steps produces no further change in the first 70 decimal digits of `sqrt(2)`. After 15 steps the approximating rational number has a numerator and a denominator with 12543 digits each, and the next step runs out of memory.

Another irrational number that is easily computed is e. The following program implements an alternating series for `1/e`, where the absolute value of last term is an upper bound on the error.

```
%
% from file: library('clpqr/examples/root')
%
e(N, E) :-
  { Err =:= exp(10,-(N+2)), Half =:= 1/2 },
  inv_e_series(Half, Half, 3, Err, Inv_E),
  { E =:= 1/Inv_E }.

inv_e_series(Term, S0, _, Err, Sum) :-
  { abs(Term) =< Err }, !,
  S0 = Sum.
inv_e_series(Term, S0, N, Err, Sum) :-
  N1 is N+1,
  { Term1 =:= -Term/N, S1 =:= Term1+S0 },
  inv_e_series(Term1, S1, N1, Err, Sum).
```

The computation of the rational number E that approximates e up to at least 1000 digits in its decimal expansion requires the evaluation of 450 terms of the series, i.e. 450 calls of `inv_e_series/5`.

```
clp(q) ?- e(1000,E).

E = 7149056228932760213666809592072842334290744221392610955845565494
    3708750229467761730471738895197792271346693089326102132000338192
    0131874187833985420922688804220167840319199699494193852403223700
    5853832741544191628747052136402176941963825543565900589161585723
    4023097417605004829991929283045372355639145644588174733401360176
    9953973706537274133283614740902771561159913069917833820285608440
    3104966899999651928637634656418969027076699082888742481392304807
    9484725489080844360397606199771786024695620205344042765860581379
    3538290451208322129898069978107971226873160872046731879753034549
    3130492167474809196348846916421782850086985668680640425192038155
    4902863298351349469211627292865440876581064873866786120098602898
    8799130098877372097360065934827751120659213470528793143805903554
    7928682131082164366007016698761961066948371407368962539467994627
    1374858249110795976398595034606994740186040425117101588480000000
```

```
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000
/
26299908104030026510959591555030022854412721706731053344668089316863103901346024240326549035084528682487048064823380723787110941
68092351873563187809723027965702511029285520037085569393147956781978390674393498540663747334079841518303636625888963910391440709
08873457973034709592078833168383469733939377783634111956243135538835644822353659840936818391050630360633734935381528275392050975
72714689928409075413503454590111924668921778668822642428604121880652112744642450404625763019639086944558899249788084559753723892
16431889914449453607268995320235429695725843637610735288411470122634218045463494055807073778490814692996517359952229262198396182
18389300435285831099738723481938068303825840405363946408951487510766256738740729894909630785260101721285704616818889741995949666
63032897031993938019763349742408153979202130597990719150678567586716458821062645562512745336709063396510021681900076680696945309
36605909332798677367479266486787385157027774313538454661996809917336187342115216547777491166010820005 9
```

The decimal expansion itself looks like this:

```
clp(q) ?- e(1000, E), print_decimal(E, 1000).
2.
7182818284 5904523536 0287471352 6624977572 4709369995 9574966967
6277240766 3035354759 4571382178 5251664274 2746639193 2003059921
8174135966 2904357290 0334295260 5956307381 3232862794 3490763233
8298807531 9525101901 1573834187 9307021540 8914993488 4167509244
7614606680 8226480016 8477411853 7423454424 3710753907 7744992069
5517027618 3860626133 1384583000 7520449338 2656029760 6737113200
7093287091 2744374704 7230696977 2093101416 9283681902 5515108657
4637721112 5238978442 5056953696 7707854499 6996794686 4454905987
9316368892 3009879312 7736178215 4249992295 7635148220 8269895193
6680331825 2886939849 6465105820 9392398294 8879332036 2509443117
3012381970 6841614039 7019837679 3206832823 7646480429 5311802328
7825098194 5581530175 6717361332 0698112509 9618188159 3041690351
5988885193 4580727386 6738589422 8792284998 9208680582 5749279610
4841984443 6346324496 8487560233 6248270419 7862320900 2160990235
3043699418 4914631409 3431738143 6405462531 5209618369 0888707016
7683964243 7814059271 4563549061 3031072085 1038375051 0115747704
1718986106 8739696552 1267154688 9570350354
```

29.5 Projection and Redundancy Elimination

Once a derivation succeeds, the Prolog system presents the bindings for the variables in the query. In a CLP system, the set of answer constraints is presented in analogy. A complication in the CLP context are variables and associated constraints that were not mentioned in the query. A motivating example is the familiar `mortgage` relation:

```
%
% from file: library('clpqr/examples/mg')
%
mg(P,T,I,B,MP):-
  {
     T = 1,
     B + MP = P * (1 + I)
  }.
mg(P,T,I,B,MP):-
  {
     T > 1,
     P1 = P * (1 + I) - MP,
     T1 = T - 1
  },
  mg(P1, T1, I, B, MP).
```

A sample query yields:

```
clp(r) ?- use_module(library('clpqr/examples/mg')).
clp(r) ?- mg(P,12,0.01,B,Mp).

{B=1.1268250301319698*P-12.682503013196973*Mp}
```

Without projection of the answer constraints onto the query variables we would observe the following interaction:

```
clp(r) ?- mg(P,12,0.01,B,Mp).

{B=12.682503013196973*_A-11.682503013196971*P},
{Mp= -(_A)+1.01*P},
{_B=2.01*_A-1.01*P},
{_C=3.0301*_A-2.0301*P},
{_D=4.060401000000001*_A-3.0604009999999997*P},
{_E=5.101005010000001*_A-4.10100501*P},
{_F=6.152015060100001*_A-5.152015060099999*P},
{_G=7.213535210701001*_A-6.213535210700999*P},
{_H=8.285670562808011*_A-7.285670562808009*P},
{_I=9.368527268436091*_A-8.36852726843609*P},
{_J=10.462212541120453*_A-9.46221254112045*P},
{_K=11.566834666531657*_A-10.566834666531655*P}
```

The variables _*A* ... _*K* are not part of the query, they originate from the mortgage program proper. Although the latter answer is equivalent to the former in terms of linear algebra, most users would prefer the former.

29.5.1 Variable Ordering

In general, there are many ways to express the same linear relationship between variables. clp(Q,R) does not care to distinguish between them, but the user might. The predicate `ordering(+`*Spec*`)` gives you some control over the variable ordering. Suppose that instead of *B*, you want *Mp* to be the defined variable:

```
clp(r) ?- mg(P,12,0.01,B,Mp).

{B=1.1268250301319698*P-12.682503013196973*Mp}
```

This is achieved with:

```
clp(r) ?-  mg(P,12,0.01,B,Mp), ordering([Mp]).

{Mp= -0.0788487886783417*B+0.08884878867834171*P}
```

One could go one step further and require *P* to appear before (to the left of) *B* in a addition:

```
clp(r) ?- mg(P,12,0.01,B,Mp), ordering([Mp,P]).

{Mp=0.08884878867834171*P-0.0788487886783417*B}
```

Spec in `ordering(+`*Spec*`)` is either a list of variables with the intended ordering, or of the form *A*`<`*B*. The latter form means that *A* goes to the left of *B*. In fact, `ordering([A,B,C,D])` is shorthand for:

```
ordering(A < B), ordering(A < C), ordering(A < D),
ordering(B < C), ordering(B < D),
ordering(C < D)
```

The ordering specification only affects the final presentation of the constraints. For all other operations of clp(Q,R), the ordering is immaterial. Note that `ordering/1` acts like a constraint: you can put it anywhere in the computation, and you can submit multiple specifications.

```
clp(r) ?- ordering(B < Mp), mg(P,12,0.01,B,Mp).

{B= -12.682503013196973*Mp+1.1268250301319698*P}

yes
clp(r) ?- ordering(B < Mp), mg(P,12,0.01,B,Mp), ordering(P < Mp).

{P=0.8874492252651537*B+11.255077473484631*Mp}
```

29.5.2 Turning Answers into Terms

In meta-programming applications one needs to get a grip on the results computed by the clp(Q,R) solver. You can use the predicates `dump/3` and/or `call_residue/2` for that purpose:

```
clp(r) ?- {2*A+B+C=10,C-D=E,A<10}, dump([A,B,C,D,E],[a,b,c,d,e],Constraints).

Constraints = [e<10.0,a=10.0-c-d-2.0*e,b=c+d],
{C=10.0-2.0*A-B},
{E=10.0-2.0*A-B-D},
{A<10.0}

clp(r) ?- call_residue({2*A+B+C=10,C-D=E,A<10}, Constraints).

Constraints = [
               [A]-{A<10.0},
               [B]-{B=10.0-2.0*A-C},
               [D]-{D=C-E}
              ]
```

29.5.3 Projecting Inequalities

As soon as linear inequations are involved, projection gets more demanding complexity wise. The current clp(Q,R) version uses a Fourier-Motzkin algorithm for the projection of linear inequalities. The choice of a suitable algorithm is somewhat dependent on the number of variables to be eliminated, the total number of variables, and other factors. It is quite easy to produce problems of moderate size where the elimination step takes some time. For example, when the dimension of the projection is 1, you might be better off computing the supremum and the infimum of the remaining variable instead of eliminating `n-1` variables via implicit projection.

In order to make answers as concise as possible, redundant constraints are removed by the system as well. In the following set of inequalities, half of them are redundant.

```
%
% from file: library('clpqr/examples/eliminat')
%
example(2, [X0,X1,X2,X3,X4]) :-
  {
       +87*X0  +52*X1  +27*X2  -54*X3  +56*X4 =<  -93,
       +33*X0  -10*X1  +61*X2  -28*X3  -29*X4 =<   63,
       -68*X0   +8*X1  +35*X2  +68*X3  +35*X4 =<  -85,
       +90*X0  +60*X1  -76*X2  -53*X3  +24*X4 =<  -68,
       -95*X0  -10*X1  +64*X2  +76*X3  -24*X4 =<   33,
       +43*X0  -22*X1  +67*X2  -68*X3  -92*X4 =<  -97,
       +39*X0   +7*X1  +62*X2  +54*X3  -26*X4 =<  -27,
       +48*X0  -13*X1   +7*X2  -61*X3  -59*X4 =<   -2,
       +49*X0  -23*X1  -31*X2  -76*X3  +27*X4 =<    3,
       -50*X0  +58*X1   -1*X2  +57*X3  +20*X4 =<    6,
```

```
        -13*X0  -63*X1  +81*X2   -3*X3  +70*X4 =<   64,
        +20*X0  +67*X1  -23*X2  -41*X3  -66*X4 =<   52,
        -81*X0  -44*X1  +19*X2  -22*X3  -73*X4 =<  -17,
        -43*X0   -9*X1  +14*X2  +27*X3  +40*X4 =<   39,
        +16*X0  +83*X1  +89*X2  +25*X3  +55*X4 =<   36,
         +2*X0  +40*X1  +65*X2  +59*X3  -32*X4 =<   13,
        -65*X0  -11*X1  +10*X2  -13*X3  +91*X4 =<   49,
        +93*X0  -73*X1  +91*X2   -1*X3  +23*X4 =<  -87
    }.
```

Consequently, the answer consists of the system of nine non-redundant inequalities only:

```
clp(q) ?- use_module(library('clpqr/examples/elimination')).
clp(q) ?- example(2, [X0,X1,X2,X3,X4]).

{X0-2/17*X1-35/68*X2-X3-35/68*X4>=5/4},
{X0-73/93*X1+91/93*X2-1/93*X3+23/93*X4=<-29/31},
{X0-29/25*X1+1/50*X2-57/50*X3-2/5*X4>=-3/25},
{X0+7/39*X1+62/39*X2+18/13*X3-2/3*X4=<-9/13},
{X0+2/19*X1-64/95*X2-4/5*X3+24/95*X4>=-33/95},
{X0+2/3*X1-38/45*X2-53/90*X3+4/15*X4=<-34/45},
{X0-23/49*X1-31/49*X2-76/49*X3+27/49*X4=<3/49},
{X0+44/81*X1-19/81*X2+22/81*X3+73/81*X4>=17/81},
{X0+9/43*X1-14/43*X2-27/43*X3-40/43*X4>=-39/43}
```

The projection (the shadow) of this polyhedral set into the `X0,X1` space can be computed via the implicit elimination of non-query variables:

```
clp(q) ?- example(2, [X0,X1|_]).

{X0+2619277/17854273*X1>=-851123/17854273},
{X0+6429953/16575801*X1=<-12749681/16575801},
{X0+19130/1213083*X1>=795400/404361},
{X0-1251619/3956679*X1>=21101146/3956679},
{X0+601502/4257189*X1>=220850/473021}
```

Projection is quite a powerful concept that leads to surprisingly terse executable specifications of nontrivial problems like the computation of the convex hull from a set of points in an n-dimensional space: Given the program

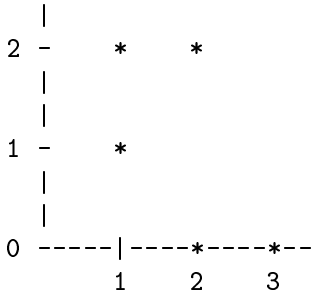
```
%
% from file: library('clpqr/examples/elimination')
%
conv_hull(Points, Xs) :-
  lin_comb(Points, Lambdas, Zero, Xs),
  zero(Zero),
  polytope(Lambdas).

polytope(Xs) :-
```

```
  positive_sum(Xs, 1).

  positive_sum([], Z) :- {Z=0}.
  positive_sum([X|Xs], SumX) :-
    { X >= 0, SumX = X+Sum },
    positive_sum(Xs, Sum).

zero([]).
zero([Z|Zs]) :- {Z=0}, zero(Zs).

lin_comb([],        [],     S1, S1).
lin_comb([Ps|Rest], [K|Ks], S1, S3) :-
  lin_comb_r(Ps, K, S1, S2),
  lin_comb(Rest, Ks, S2, S3).

  lin_comb_r([],     _, [],     []).
  lin_comb_r([P|Ps], K, [S|Ss], [Kps|Ss1]) :-
    { Kps = K*P+S },
    lin_comb_r(Ps, K, Ss, Ss1).
```

we can post the following query:

```
clp(q) ?- conv_hull([ [1,1], [2,0], [3,0], [1,2], [2,2] ], [X,Y]).

{Y=<2},
{X+1/2*Y=<3},
{X>=1},
{Y>=0},
{X+Y>=2}
```

This answer is easily verified graphically:



The convex hull program directly corresponds to the mathematical definition of the convex hull. What does the trick in operational terms is the implicit elimination of the *Lambdas* from the program formulation. Please note that this program does not limit the number of points or the dimension of the space they are from. Please note further that quantifier elimination is a computationally expensive operation and therefore this program is only useful as a benchmark for the projector and not so for the intended purpose.

29.6 Why Disequations

A beautiful example of disequations at work is due to [Colmerauer 90]. It addresses the task of tiling a rectangle with squares of all-different, a priori unknown sizes. Here is a translation of the original `Prolog-III` program to clp(Q,R):

```
%
% from file: library('clpqr/examples/squares')
%
filled_rectangle(A, C) :-
  { A >= 1 },
  distinct_squares(C),
  filled_zone([-1,A,1], _, C, []).

distinct_squares([]).
distinct_squares([B|C]) :-
  { B > 0 },
  outof(C, B),
  distinct_squares(C).

outof([],     _).
outof([B1|C], B) :-
  { B =\= B1 },       % *** note disequation ***
  outof(C, B).

filled_zone([V|L], [W|L], C0, C0) :-
  { V=W,V >= 0 }.
filled_zone([V|L], L3, [B|C], C2) :-
  { V < 0 },
  placed_square(B, L, L1),
  filled_zone(L1, L2, C, C1),
  { Vb=V+B },
  filled_zone([Vb,B|L2], L3, C1, C2).

placed_square(B, [H,H0,H1|L], L1) :-
  { B > H, H0=0, H2=H+H1 },
  placed_square(B, [H2|L], L1).
placed_square(B, [B,V|L], [X|L]) :-
  { X=V-B }.
placed_square(B, [H|L], [X,Y|L]) :-
  { B < H, X= -B, Y=H-B }.
```

There are no tilings with less than nine squares except the trivial one where the rectangle equals the only square. There are eight solutions for nine squares. Six further solutions are rotations of the first two.

```
clp(q) ?- use_module(library('clpqr/examples/squares')).
clp(q) ?- filled_rectangle(A, Squares).

A = 1,
Squares = [1] ? ;

A = 33/32,
Squares = [15/32,9/16,1/4,7/32,1/8,7/16,1/32,5/16,9/32] ? ;

A = 69/61,
Squares = [33/61,36/61,28/61,5/61,2/61,9/61,25/61,7/61,16/61]
```

Depending on your hardware, the above query may take a few minutes. Supplying the knowledge about the minimal number of squares beforehand cuts the computation time by a factor of roughly four:

```
clp(q) ?- length(Squares, 9), filled_rectangle(A, Squares).

A = 33/32,
Squares = [15/32,9/16,1/4,7/32,1/8,7/16,1/32,5/16,9/32] ? ;

A = 69/61,
Squares = [33/61,36/61,28/61,5/61,2/61,9/61,25/61,7/61,16/61]
```

29.7 Syntactic Sugar

There is a package that transforms programs and queries from a eval-quote variant of clp(Q,R) into corresponding programs and queries in a quote-eval variant. Before you use it, you need to know that in an eval-quote language, all symbols are interpreted unless explicitly quoted. This means that interpreted terms cannot be manipulated syntactically directly. Meta-programming in a CLP context by definition manipulates interpreted terms, therefore you need `quote/1` (just as in LISP) and some means to put syntactical terms back to their interpreted life: `{}/1`.

In a quote-eval language, meta-programming is (pragmatically) simpler because everything is implicitly quoted until explicitly evaluated. On the other hand, now object programming suffers from the dual inconvenience.

We chose to make our version of clp(Q,R) of the quote-eval type because this matches the intended use of the already existing boolean solver of SICStus. In order to keep the users of the eval-quote variant happy, we provide a source transformation package. It is activated via:

```
| ?- use_module(library('clpqr/expand')).
```

Loading the package puts you in a mode where the arithmetic functors like `+/2`, `*/2` and all numbers (functors of arity 0) are interpreted semantically.

```
clp(r) ?- 2+2=X.

X = 4.0
```

The package works by *purifying* programs and queries in the sense that all references to interpreted terms are made explicit. The above query is expanded prior to evaluation into:

```
{2.0+2.0=X}
```

The same mechanism applies when interpreted terms are nested deeper:

```
some_predicate(10, f(A+B/2), 2*cos(A))
```

Expands into:

```
{Xc=2.0*cos(A)},
{Xb=A+B/2},
{Xa=10.0},
some_predicate(Xa, f(Xb), Xc)
```

This process also applies when files are consulted or compiled. In fact, this is the only situation where expansion can be applied with relative safety. To see this, consider what happens when the top level evaluates the expansion, namely some calls to the clp(Q,R) solver, followed by the call of the purified query. As we learned in Section 29.2.3 [Feedback], page 253, the solver may bind variables, which produces a goal with interpreted functors in it (numbers), which leads to another stage of expansion, and so on.

We recommend that you only turn on expansion temporarily while consulting or compiling files needing expansion with `expand/0` and `noexpand/0`.

29.7.1 Monash Examples

This collection of examples has been distributed with the Monash University Version of clp(R) [Heintze et al. 87], and its inclusion into this distribution was kindly permitted by Roland Yap.

In order to execute the examples, a small compatibility package has to be loaded first:

```
clp(r) ?- use_module(library('clpqr/monash')).
```

Then, assuming you are using clp(R):

```
clp(r) ?- expand, [library('clpqr/examples/monash/rkf45')],
          noexpand.

clp(r) ?- go.
Point    0.00000 :    0.75000    0.00000
Point    0.50000 :    0.61969    0.47793
Point    1.00000 :    0.29417    0.81233
Point    1.50000 :   -0.10556    0.95809
```

```
Point    2.00000 :   -0.49076    0.93977
Point    2.50000 :   -0.81440    0.79929
Point    3.00000 :   -1.05440    0.57522

Iteration finished
------------------
 439  derivative evaluations
```

29.7.1.1 Compatibility Notes

The Monash examples have been written for clp(R). Nevertheless, all but `rkf45` complete nicely in clp(Q). With `rkf45`, clp(Q) runs out of memory. This is an instance of the problem discussed in Section 29.4 [Numerical Precision], page 255.

The Monash University clp(R) interpreter features a `dump/n` predicate. It is used to print the target variables according to the given ordering. Within this version of clp(Q,R), the corresponding functionality is provided via `ordering/1`. The difference is that `ordering/1` does only specify the ordering of the variables and *no* printing is performed. We think Prolog has enough predicates to perform output already. You can still run the examples referring to `dump/n` from the Prolog top level:

```
clp(r) ?- expand, [library('clpqr/examples/monash/mortgage')], noexpand.

% go2
%
clp(r) ?- mg(P,120,0.01,0,MP), dump([P,MP]).

{P=69.7005220313972*MP}

% go3
%
clp(r) ?- mg(P,120,0.01,B,MP), dump([P,B,MP]).

{P=0.30299477968602706*B+69.7005220313972*MP}

% go4
%
clp(r) ?- mg(999, 3, Int, 0, 400), dump.

clpr:{_B-_B*Int+_A+400.0=0.0},
clpr:{_A-_A*Int+400.0=0.0},
{_B=599.0+999.0*Int}
```

29.8 A Mixed Integer Linear Optimization Example

The predicates `bb_inf/3, bb_inf/5` implement a simple Branch and Bound search algorithm for Mixed Integer Linear (MIP) Optimization examples. Serious MIP is not trivial. The implementation `library('clpqr/bb.pl')` is to be understood as a starting point for more ambitious users who need control over branching, or who want to add cutting planes, for example.

Anyway, here is a small problem from miplib, a collection of MIP models, housed at Rice University:

```
NAME:          flugpl
ROWS:          18
COLUMNS:       18
INTEGER:       11
NONZERO:       46
BEST SOLN:     1201500 (opt)
LP SOLN:       1167185.73
SOURCE:        Harvey M. Wagner
               John W. Gregory (Cray Research)
               E. Andrew Boyd (Rice University)
APPLICATION:   airline model
COMMENTS:      no integer variables are binary

%
% from file: library('clpqr/examples/mip')
%
example(flugpl, Obj, Vs, Ints, []) :-
  Vs = [ Anm1,Anm2,Anm3,Anm4,Anm5,Anm6,
         Stm1,Stm2,Stm3,Stm4,Stm5,Stm6,
         UE1,UE2,UE3,UE4,UE5,UE6],
  Ints = [Stm6, Stm5, Stm4, Stm3, Stm2,
          Anm6, Anm5, Anm4, Anm3, Anm2, Anm1],

  Obj =    2700*Stm1 + 1500*Anm1 + 30*UE1
         + 2700*Stm2 + 1500*Anm2 + 30*UE2
         + 2700*Stm3 + 1500*Anm3 + 30*UE3
         + 2700*Stm4 + 1500*Anm4 + 30*UE4
         + 2700*Stm5 + 1500*Anm5 + 30*UE5
         + 2700*Stm6 + 1500*Anm6 + 30*UE6,

  allpos(Vs),
  {  Stm1 = 60, 0.9*Stm1 +1*Anm1 -1*Stm2 = 0,
     0.9*Stm2 +1*Anm2 -1*Stm3 = 0, 0.9*Stm3 +1*Anm3 -1*Stm4 = 0,
     0.9*Stm4 +1*Anm4 -1*Stm5 = 0, 0.9*Stm5 +1*Anm5 -1*Stm6 = 0,
     150*Stm1 -100*Anm1 +1*UE1 >= 8000,
     150*Stm2 -100*Anm2 +1*UE2 >= 9000,
     150*Stm3 -100*Anm3 +1*UE3 >= 8000,
     150*Stm4 -100*Anm4 +1*UE4 >= 10000,
     150*Stm5 -100*Anm5 +1*UE5 >= 9000,
     150*Stm6 -100*Anm6 +1*UE6 >= 12000,
```

```
    -20*Stm1 +1*UE1 =< 0, -20*Stm2 +1*UE2 =< 0, -20*Stm3 +1*UE3 =< 0,
    -20*Stm4 +1*UE4 =< 0, -20*Stm5 +1*UE5 =< 0, -20*Stm6 +1*UE6 =< 0,
    Anm1 =< 18, 57 =< Stm2, Stm2 =< 75, Anm2 =< 18,
    57 =< Stm3, Stm3 =< 75, Anm3 =< 18, 57 =< Stm4,
    Stm4 =< 75, Anm4 =< 18, 57 =< Stm5, Stm5 =< 75,
    Anm5 =< 18, 57 =< Stm6, Stm6 =< 75, Anm6 =< 18
  }.

allpos([]).
allpos([X|Xs]) :- {X >= 0}, allpos(Xs).
```

We can first check whether the relaxed problem has indeed the quoted infimum:

```
clp(r) ?- example(flugpl, Obj, _, _, _), inf(Obj, Inf).

Inf = 1167185.7255923203
```

Computing the infimum under the additional constraints that `Stm6`, `Stm5`, `Stm4`, `Stm3`, `Stm2`, `Anm6`, `Anm5`, `Anm4`, `Anm3`, `Anm2`, `Anm1` assume integer values at the infimum is computationally harder, but the query does not change much:

```
clp(r) ?- example(flugpl, Obj, _, Ints, _),
          bb_inf(Ints, Obj, Inf, Vertex, 0.001).

Inf = 1201500.0000000005,
Vertex = [75.0,70.0,70.0,60.0,60.0,0.0,12.0,7.0,16.0,6.0,6.0]
```

29.9 Implementation Architecture

The system consists roughly of the following components:

- A polynomial normal form expression simplification mechanism.
- A solver for linear equations [Holzbaur 92].
- A simplex algorithm to decide linear inequalities [Holzbaur 94].

29.9.1 Fragments and Bits

29.9.1.1 Rationals

The internal data structure for rational numbers is `rat(Num,Den)`. *Den* is always positive, i.e. the sign of the rational number is the sign of *Num*. Further, *Num* and *Den* are relative prime. Note that integer *N* looks like `rat(N,1)` in this representation. You can control printing of terms with `portray/1`.

29.9.1.2 Partial Evaluation, Compilation

Once one has a working solver, it is obvious and attractive to run the constraints in a clause definition at read time or compile time and proceed with the answer constraints in place of the original constraints. This gets you constant folding and in fact the full algebraic power of the solver applied to the avoidance of computations at runtime. The mechanism to realize this idea is to use `dump/3, call_residue/2` for the expansion of `{}/1`, via hook predicate `user:goal_expansion/3`).

29.9.1.3 Asserting with Constraints

If you use the dynamic data base, the clauses you assert might have constraints on the variables occurring in the clause. This works as expected:

```
clp(r) ?- {A < 10}, assert(p(A)).

{A<10.0}

yes
clp(r) ?- p(X).

{X<10.0}
```

29.9.2 Bugs

- The fuzzy comparison of floats is the source for all sorts of weirdness. If a result in R surprises you, try to run the program in Q before you send me a bug report.
- The projector for floundered nonlinear relations keeps too many variables. Its output is rather unreadable.
- Disequations are not projected properly.
- This list is probably incomplete.

Please send bug reports to `<christian@ai.univie.ac.at>`.

30 Constraint Logic Programming over Finite Domains

30.1 Introduction

The clp(FD) solver described in this chapter is an instance of the general Constraint Logic Programming scheme introduced in [Jaffar & Michaylov 87]. This constraint domain is particularly useful for modeling discrete optimization and verification problems such as scheduling, planning, packing, timetabling etc. The treatise [Van Hentenryck 89] is an excellent exposition of the theoretical and practical framework behind constraint solving in finite domains, and summarizes the work up to 1989.

This solver has the following highlights:

- Two classes of constraints are handled internally: primitive constraints and global constraints.
- The constraints described in this chapter are automatically translated to conjunctions of primitive and global library constraints.
- The truth value of a primitive constraint can be reflected into a 0/1-variable (reification).
- New primitive constraints can be added by writing so-called indexicals.
- New global constraints can be written in Prolog, by means of a programming interface.

The rest of this chapter is organized as follows: How to load the solver and how to write simple programs is explained in Section 30.2 [CLPFD Interface], page 272. A description of all constraints that the solver provides is contained in Section 30.3 [Available Constraints], page 275. The predicates for searching for solution are documented in Section 30.4 [Enumeration Predicates], page 280. The predicates for getting execution statistics are documented in Section 30.5 [Statistics Predicates], page 282. A few example programs are given in Section 30.10 [Example Programs], page 298. Finally, Section 30.11 [Syntax Summary], page 301 contains syntax rules for all expressions.

The following sections discuss advanced features and are probably only relevant to experienced users: How to control the amount of information presented in answers to queries is explained in Section 30.6 [Answer Constraints], page 283. The solver's execution mechanism and primitives are described in Section 30.7 [The Constraint System], page 283. How to add new global constraints via a programming interface is described in Section 30.8 [Defining Global Constraints], page 285. How to define new primitive constraints with indexicals is described in Section 30.9 [Defining Primitive Constraints], page 290.

Referencing this Software

When referring to this implementation of clp(FD) in publications, please use the following reference:

> Carlsson M., Ottosson G., Carlson B. "An Open-Ended Finite Domain Constraint Solver" Proc. Programming Languages: Implementations, Logics, and Programs, 1997.

Acknowledgments

The first version of this solver was written as part of Key Hyckenberg's MSc thesis in 1995. The code was later rewritten by Mats Carlsson, with contributions from Greger Ottosson at the Computing Science Department, Uppsala University. Peter Szeredi contributed material for this manual chapter.

The development of this software was supported by the Swedish National Board for Technical and Industrial Development (NUTEK) under the auspices of Advanced Software Technology (ASTEC) Center of Competence at Uppsala University.

We include a collection of examples, some of which have been distributed with the INRIA implementation of clp(FD) [Diaz & Codognet 93].

30.2 Solver Interface

The solver is available as a library module and can be loaded with a query

```
:- use_module(library(clpfd)).
```

The solver contains predicates for checking the consistency and entailment of finite domain constraints, as well as solving for solution values for your problem variables.

In the context of this constraint solver, a *finite domain* is a subset of the integers, and a *finite domain constraint* denotes a relation over one or more finite domains. All *domain variables*, i.e. variables that occur as arguments to finite domain constraints get associated with a finite domain, either explicitly declared by the program, or implicitly imposed by the constraint solver. Temporarily, the domain of a variable may actually be infinite, if it does not have a finite lower or upper bound.

The domain of all variables gets narrower and narrower as more constraints are added. If a domain becomes empty, the accumulated constraints are unsatisfiable, and the current computation branch fails. At the end of a successful computation, all domains have usually become singletons, i.e. the domain variables have become assigned.

The domains do not become singletons automatically. Usually, it takes some amount of search to find an assignment that satisfies all constraints. It is the programmer's responsibility to do so. If some domain variables are left unassigned in a computation, the garbage collector will preserve all constraint data that is attached to them. Furthermore, the behavior of the predicates `assert/1`, `findall/3`, `copy_term/2` and friends is undefined on non-ground terms containing domain variables.

The heart of the constraint solver is a scheduler for indexicals [Van Hentenryck et al. 92] and global constraints. Both entities act as coroutines performing incremental constraint solving or entailment checking. They wake up by changes in the domains of its arguments. All constraints provided by this package are implemented as indexicals or global constraints. New constraints can be defined by the user.

Indexicals are reactive functional rules that take part in the solver's basic constraint solving algorithm, whereas each global constraint is associated with its particular constraint solving algorithm. The solver maintains two scheduling queues, giving priority to the queue of indexicals.

The feasibility of integrating the indexical approach with a Prolog based on the WAM was clearly demonstrated by Diaz's clp(FD) implementation [Diaz & Codognet 93], one of the fastest finite domains solvers around.

30.2.1 Posting Constraints

A constraint is called as any other Prolog predicate. When called, the constraint is *posted* to the store. For example:

```
| ?- X in 1..5, Y in 2..8, X+Y #= T.

X in 1..5,
Y in 2..8,
T in 3..13 ?

yes
| ?- X in 1..5, T in 3..13, X+Y #= T.

X in 1..5,
T in 3..13,
Y in -2..12 ?

yes
```

Note that the answer constraint shows the domains of nonground query variables, but not any constraints that may be attached to them.

30.2.2 A Constraint Satisfaction Problem

Constraint satisfaction problems (CSPs) are a major class of problems for which this solver is ideally suited. In a CSP, the goal is to pick values from pre-defined domains for certain variables so that the given constraints on the variables are all satisfied.

As a simple CSP example, let us consider the Send More Money puzzle. In this problem, the variables are the letters S, E, N, D, M, O, R, and Y. Each letter represents a digit between 0 and 9. The problem is to assign a value to each digit, such that SEND + MORE equals MONEY.

A program which solves the puzzle is given below. The program contains the typical three steps of a clp(FD) program:

1. declare the domains of the variables
2. post the problem constraints

3. look for a feasible solution via backtrack search, or look for an optimal solution via branch-and-bound search

Sometimes, an extra step precedes the search for a solution: the posting of surrogate constraints to break symmetries or to otherwise help prune the search space. No surrogate constraints are used in this example.

The domains of this puzzle are stated via the `domain/3` goal and by requiring that S and M be greater than zero. The two problem constraint of this puzzle are the equation (`sum/8`) and the constraint that all letters take distinct values (`all_different/1`). Finally, the backtrack search is performed by `labeling/2`. Different search strategies can be encoded in the `Type` parameter. In the example query, the default search strategy is used (select the leftmost variable, try values in ascending order).

```
:- use_module(library(clpfd)).

mm([S,E,N,D,M,O,R,Y], Type) :-
     domain([S,E,N,D,M,O,R,Y], 0, 9),      % step 1
     S#>0, M#>0,
     all_different([S,E,N,D,M,O,R,Y]),     % step 2
     sum(S,E,N,D,M,O,R,Y),
     labeling(Type, [S,E,N,D,M,O,R,Y]).    % step 3

sum(S, E, N, D, M, O, R, Y) :-
                   1000*S + 100*E + 10*N + D
     +             1000*M + 100*O + 10*R + E
     #= 10000*M + 1000*O + 100*N + 10*E + Y.

| ?- mm([S,E,N,D,M,O,R,Y], []).

D = 7,
E = 5,
M = 1,
N = 6,
O = 0,
R = 8,
S = 9,
Y = 2 ?
```

30.2.3 Reified Constraints

Instead of merely posting constraints it is often useful to reflect its truth value into a 0/1-variable B, so that:

- the constraint is posted if B is set to 1
- the negation of the constraint is posted if B is set to 0
- B is set to 1 if the constraint becomes entailed

- B is set to 0 if the constraint becomes disentailed

This mechanism is known as *reification*. Several frequently used operations can be defined in terms of reified constraints, such as blocking implication [Saraswat 90] and the cardinality operator [Van Hentenryck & Deville 91], to name a few. A reified constraint is written:

```
| ?- Constraint #<=> B.
```

where *Constraint* is reifiable. As an example of a constraint that uses reification, consider `exactly(X,L,N)` which is true if `X` occurs exactly `N` times in the list `L`. It can be defined thus:

```
exactly(_, [], 0).
exactly(X, [Y|L], N) :-
    X #= Y #<=> B,
    N #= M+B,
    exactly(X, L, M).
```

30.3 Available Constraints

This section describes the classes of constraints that can be used with this solver.

30.3.1 Arithmetic Constraints

?Expr RelOp ?Expr

defines an arithmetic constraint. The syntax for *Expr* and *RelOp* is defined by a grammar (see [Syntax of Arithmetic Expressions], page 303). Note that the expressions are not restricted to being linear. Constraints over non-linear expressions, however, will usually yield less constraint propagation than constraints over linear expressions.

Arithmetic constraints can be reified as e.g.

```
| ?- X in 1..2, Y in 3..5, X#=<Y #<=> B.

B = 1,
X in 1..2,
Y in 3..5 ?
```

Linear arithmetic constraints maintain (at least) interval-consistency and their reified versions detect (at least) interval-entailment and -disentailment; see Section 30.7 [The Constraint System], page 283.

The following constraints are among the library constraints that general arithmetic constraints compile to. They express a relation between a sum or a scalar product and a value, using a dedicated algorithm that avoids creating any temporary variables holding intermediate values. If you are computing a sum or a scalar product, it can be much more efficient to compute lists of coefficients and variables and post a single sum or scalar product constraint than to post a sequence of elementary constraints.

`sum(+`*Xs*`, +`*RelOp*`, ?`*Value*`)`
: where *Xs* is a list of integers or domain variables, *RelOp* is a relational symbol as above, and *Value* is an integer or a domain variable. True if *Xs RelOp Value*. Cannot be reified.

`scalar_product(+`*Coeffs*`, +`*Xs*`, +`*RelOp*`, ?`*Value*`)`
: where *Coeffs* is a list of length *n* of integers, *Xs* is a list of length *n* of integers or domain variables, *RelOp* is a relational symbol as above, and *Value* is an integer or a domain variable. True if *Coeffs*Xs RelOp Value*. Cannot be reified.

30.3.2 Membership Constraints

`domain(+`*Variables*`, +`*Min*`, +`*Max*`)`
: where *Variables* is a list of domain variables or integers, *Min* is an integer or the atom `inf` (minus infinity), and *Max* is an integer or the atom `sup` (plus infinity). True if the variables all are elements of the range *Min*`..`*Max*. Cannot be reified.

?X `in +`*Range*
: defines a membership constraint. *X* is an integer or a domain variable and *Range* is a *ConstantRange* (see [Syntax of Indexicals], page 301). True if *X* is an element of the range.

?X `in_set +`*FDSet*
: defines a membership constraint. *X* is an integer or a domain variable and *FDSet* is an FD set term (see Section 30.8.3 [FD Set Operations], page 287). True if *X* is an element of the FD set.

`in/2` and `in_set/2` constraints can be reified. They maintain domain-consistency and their reified versions detect domain-entailment and -disentailment; see Section 30.7 [The Constraint System], page 283.

30.3.3 Propositional Constraints

Propositional combinators can be used to combine reifiable constraints into propositional formulae over such constraints. Such formulae are goal expanded by the system into sequences of reified constraints and arithmetic constraints. For example,

```
X #= 4 #\/ Y #= 6
```

expresses the disjunction of two equality constraints.

The leaves of propositional formulae can be reifiable constraints, the constants 0 and 1, or 0/1-variables. New primitive, reifiable constraints can be defined with indexicals as described in Section 30.9 [Defining Primitive Constraints], page 290. The following propositional combinators are available:

`#\` *:Q*
: True if the constraint *Q* is false.

:P `#/\` *:Q* True if the constraints P and Q are both true.

:P `#\` *:Q* True if exactly one of the constraints P and Q is true.

:P `#\/` *:Q* True if at least one of the constraints P and Q is true.

:P `#=>` *:Q*
:Q `#<=` *:P* True if the constraint Q is true or the constraint P is false.

:P `#<=>` *:Q*
True if the constraints P and Q are both true or both false.

Note that the reification scheme introduced in Section 30.2.3 [Reified Constraints], page 274 is a special case of a propositional constraint.

30.3.4 Combinatorial Constraints

The constraints listed here are sometimes called symbolic constraints. They are currently not reifiable.

`count(+`*Val*`,+`*List*`,+`*RelOp*`,?`*Count*`)`
where *Val* is an integer, *List* is a list of integers or domain variables, *Count* an integer or a domain variable, and *RelOp* is a relational symbol as in Section 30.3.1 [Arithmetic Constraints], page 275. True if N is the number of elements of *List* that are equal to *Val* and N *RelOp Count*. Thus, `count/4` is a generalization of `exactly/3` (not an exported constraint) that was used in an example earlier.

`count/4` maintains domain-consistency; see Section 30.7 [The Constraint System], page 283.

`element(?`X`,+`*List*`,?`Y`)`
where X and Y are integers or domain variables and *List* is a list of integers or domain variables. True if the X:th element of *List* is Y. Operationally, the domains of X and Y are constrained so that for every element in the domain of X, there is a compatible element in the domain of Y, and vice versa.

This constraint uses an optimized algorithm for the special case where *List* is ground.

`element/3` maintains domain-consistency; see Section 30.7 [The Constraint System], page 283.

`relation(?`X`,+`*MapList*`,?`Y`)`
where X and Y are integers or domain variables and *MapList* is a list of *integer*`-`*ConstantRange* pairs, where the integer keys occur uniquely (see [Syntax of Indexicals], page 301). True if *MapList* contains a pair X`-`R and Y is in the range denoted by R.

Operationally, the domains of X and Y are constrained so that for every element in the domain of X, there is a compatible element in the domain of Y, and vice versa.

If *MapList* is not ground, the constraint must be wrapped in `call/1` to postpone goal expansion until runtime.

An arbitrary binary constraint can be defined with `relation/3`. `relation/3` maintains domain-consistency; see Section 30.7 [The Constraint System], page 283.

`all_different(+`*Variables*`)`
: where *Variables* is a list of domain variables or integers. Each variable is constrained to take a value that is unique among the variables. Operationally, this is equivalent to posting an inequality constraint for each pair of variables.

  This constraint wakes up each time a variable becomes ground, pruning the domains of the other variables. The implementation is incomplete, i.e. domain elements for *Variables* for which the constraint has no solutions are not always eliminated. A complete version is provided by the `all_distinct/1` constraint (see below).

The following constraint is a complete version of the `all_different/1` constraint. The algorithm is due to Regin [Regin 94].

`all_distinct(+`*Variables*`)`
: where *Variables* is a list of domain variables or integers. Each variable is constrained to take a value that is unique among the variables. The implementation is complete, i.e. it maintains domain-consistency; see Section 30.7 [The Constraint System], page 283.

The following is a constraint over two lists of length *n* of variables. Each variable is constrained to take a value in 1,...,*n* that is unique for its list. Furthermore, the lists are dual in a sense described below.

`assignment(+`*Xs*`, +`*Ys*`)`
: where *Xs* and *Ys* are lists of domain variables or integers, both of length *n*. True if all *Xi*, *Yi* in 1,...,*n* and *Xi*=*j* iff *Yj*=*i*.

The following constraint can be thought of as constraining *n* nodes in a graph to form a Hamiltonian circuit. The nodes are numbered from 1 to *n*. The circuit starts in node 1, visits each node, and returns to the origin.

`circuit(+`*Succ*`)`
`circuit(+`*Succ*`, +`*Pred*`)`
: where *Succ* is a list of length *n* of domain variables or integers. The *i*:th element of *Succ* (*Pred*) is the successor (predecessor) of *i* in the graph. True if the values form a Hamiltonian circuit.

The following four constraints can be thought of as constraining *n* tasks, each with a start time *Sj* and a duration *Dj*, so that no tasks ever overlap. The tasks can be seen as competing for some exclusive resource.

`serialized(+`*Starts*`,+`*Durations*`)`
: where *Starts* = [*S1*,...,*Sn*] and *Durations* = [*D1*,...,*Dn*] are lists of domain variables with finite bounds or integers. True if:

  Si`+`*Di* `=<` *Sj* `OR` *Sj*`+`*Dj* `=<` *Si*, for all `1 =<` *i*`<`*j* `=<` *n*

  The `serialized/2` constraint is merely a special case of `cumulative/4` (see below).

`serialized_resource(+`*Starts*`,+`*Durations*`,-`*Resource*`)`
: expresses the same constraint, returning in *Resource* a term which can be passed to `order_resource/2` (see Section 30.4 [Enumeration Predicates], page 280) in order to find a consistent ordering of the tasks.

`serialized_precedence(+`*Starts*`,+`*Durations*`,+`*Precedences*`)`
`serialized_precedence_resource(+`*Starts*`,+`*Durations*`,+`*Precedences*`,-`*Resource*`)`
: express the same constraint, combined with some extra precedence constraint as encoded in *Precedences* which should be a list of terms of the form `d(`*i*`,`*j*`,`*d*`)`. Here, *i* and *j* should be task numbers, and *d* should be a positive integer or `sup`. Each term adds the constraint:

  Si`+`*d* `=<` *Sj* `OR` *Sj* `=<` *Si*, if *d* is an integer
  Sj `=<` *Si*, if *d* is `sup`

  `serialized_precedence/3` can model a set of tasks to be serialized with sequence-dependent setup times. For example, the following constraint models three tasks, all with duration 5, where task 1 must precede task 2 and task 3 must either complete before task 2 or start at least 10 time units after task 2 started:

```
?- domain([S1,S2,S3], 0, 20),
   serialized_precedence([S1,S2,S3], [5,5,5], [d(2,1,sup),d(2,3,10)]).

S1 in 0..15,
S2 in 5..20,
S3 in 0..20 ?
```

  The bounds of `S1` and `S2` changed because of the precedence constraint. Setting `S2` to 5 will propagate `S1=0` and `S3 in 15..20`.

The following constraint can be thought of as constraining *n* tasks, each with a start time *Sj*, a duration *Dj*, and a resource amount *Rj*, so that the total resource consumption does not exceed *Limit* at any time:

`cumulative(+`*Starts*`,+`*Durations*`,+`*Resources*`,?`*Limit*`)`
: where *Starts* = [*S1*,...,*Sn*], *Durations* = [*D1*,...,*Dn*], *Resource* = [*R1*,...,*Rn*] are lists of domain variables with finite bounds or integers, and *Limit* is a domain variable with finite bounds or an integer. Let

  a `= min(`*S1*`,...,`*Sn*`)`,
  b `= max(`*S1*`+`*D1*`,...,`*Sn*`+`*Dn*`)`
  Rij `= Rj, if` *Sj* `=<` *i* `<` *Sj*`+`*Dj*
  Rij `= 0` otherwise

  The constraint holds if:

  Ri1`+...+`*Rin* `=<` *Limit*, for all *a* `=<` *i* `<` *b*

  The `cumulative/4` constraint is due to Aggoun and Beldiceanu [Aggoun & Beldiceanu 92]. The current implementation is incomplete and takes *O(n^2)* space.

30.3.5 User-Defined Constraints

New, primitive constraints can be added defined by the user on two different levels. On a higher level, constraints can be defined using the global constraint programming interface; see Section 30.8 [Defining Global Constraints], page 285. Such constraints can embody specialized algorithms and use the full power of Prolog. They cannot be reified.

On a lower level, new primitive constraints can be defined with indexicals. In this case, they take part in the basic constraint solving algorithm and express custom designed rules for special cases of the overall local propagation scheme. Such constraints are called *FD predicates*; see Section 30.9 [Defining Primitive Constraints], page 290. They can optionally be reified.

30.4 Enumeration Predicates

As is usually the case with finite domain constraint solvers, this solver is not *complete*. That is, it does not ensure that the set of posted constraints is satisfiable. One must resort to search (enumeration) to check satisfiability and get particular solutions.

The following predicates provide several variants of search:

`indomain(`*?X*`)`
: where X is a domain variable with finite bounds or an integer. Assigns, in increasing order via backtracking, a feasible value to X.

`minimize(`*:Goal*`,`*?X*`)`
`maximize(`*:Goal*`,`*?X*`)`
: Uses a branch-and-bound algorithm with restart to find an assignment that minimizes (maximizes) the domain variable X. *Goal* should be a Prolog goal that constrains X to become assigned, and could be a `labeling/2` goal. The algorithm calls *Goal* repeatedly with a progressively tighter upper (lower) bound on X until a proof of optimality is obtained, at which time *Goal* and X are unified with values corresponding to the optimal solution.

`labeling(+`*Options*`, +`*Variables*`)`
: where *Variables* is a list of domain variables or integers and *Options* is a list of search options. True if an assignment of the variables can be found which satisfies the posted constraints.

The search options control the order in which variables are selected for assignment (variable choice heuristic), the way in which choices are made for the selected variable (value choice heuristic), and whether all solutions or a single, optimal solution should be found. The search options are divided into four groups. One option may be selected per group. Finally, the number of assumptions (choices) made during the search can be collected.

> The following options control the order in which the next variable is selected for assignment. None of these methods will select a variable with an unbounded domain. Thus `labeling/2` may succeed with such variables still unassigned.

`leftmost` The leftmost variable is selected. This is the default.

`min` The leftmost variable with the smallest lower bound is selected.

`max` The leftmost variable with the greatest upper bound is selected.

`ff` The first-fail principle is used: the leftmost variable with the smallest domain is selected.

`ffc` The most constrained heuristic is used: a variable with the smallest domain is selected, breaking ties by (a) selecting the variable that has the most constraints suspended on it and (b) selecting the leftmost one.

The following options control the way in which choices are made for the selected variable X:

`step` Makes a binary choice between X `#=` B and X `#\=` B, where B is the lower or upper bound of X. This is the default.

`enum` Makes a multiple choice for X corresponding to the values in its domain.

`bisect` Makes a binary choice between X `#=<` M and X `#>` M, where M is the midpoint of the domain of X. This strategy is also known as domain splitting.

The following options control the order in which the choices are made for the selected variable X:

`up` The domain is explored in ascending order. This is the default.

`down` The domain is explored in descending order.

The following options control whether all solutions should be enumerated by backtracking or whether a single solution that minimizes (maximizes) X is returned, if one exists.

`all` All solutions are enumerated. This is the default.

`minimize(`X`)`
`maximize(`X`)`
Uses a branch-and-bound algorithm to find an assignment that minimizes (maximizes) the domain variable X. The labeling should constrain X to become assigned for all assignments of *Variables*.

Finally, the following option counts the number of assumptions (choices) made during the search:

`statistics(`K`)`
When a solution is found, K is unified with the number of choices made.

For example, to enumerate solutions using a static variable ordering, use:

```
| ?- constraints(Variables),
     labeling([], Variables).
     %same as [leftmost,step,up,all]
```

To minimize a cost function using branch-and-bound search, a dynamic variable ordering using the first-fail principle, and domain splitting exploring the upper part of domains first, use:

```
| ?- constraints(Variables, Cost),
     labeling([ff,bisect,down,minimize(Cost)], Variables).
```

As opposed to the predicates above which search for consistent assignments to domain variables, the following predicate searches for a consistent ordering among tasks competing for an exclusive resource, without necessarily fixing their start times:

`order_resource(+`*Options*`, +`*Resource*`)`

where *Options* is a list of search options and *Resource* represents a resource as returned by `serialized_resource/3` (see Section 30.3.4 [Combinatorial Constraints], page 277) on which tasks must be serialized. True if a total ordering can be imposed on the tasks, enumerating all such orderings via backtracking.

The search options control the construction of the total ordering. It may contain at most one of the following atoms, selecting a strategy:

`first` The ordering is built by repetitively selecting some task to be placed before all others.

`last` The ordering is built by repetitively selecting some task to be placed after all others.

and at most one of the following atoms, controlling which task to select at each step. If `first` is chosen (the default), the task with the smallest value is selected, otherwise the task with the greatest value is selected.

`est` The tasks are ordered by earliest start time.

`lst` The tasks are ordered by latest start time.

`ect` The tasks are ordered by earliest completion time.

`lct` The tasks are ordered by latest completion time.

`[first,est]` (the default) and `[last,lct]` can be good heuristics.

30.5 Statistics Predicates

The following predicates can be used to get execution statistics.

`fd_statistics(?`*Key*`, ?`*Value*`)`

This allows a program to access execution statistics specific to this solver. General statistics about CPU time and memory consumption etc. is available from the built-in predicate `statistics/2`.

For each of the possible keys *Key*, *Value* is unified with the current value of a counter which is simultaneously zeroed. The following counters are maintained. See Section 30.7 [The Constraint System], page 283, for details of what they all mean:

`resumptions`
The number of times a constraint was resumed.

`entailments`
: The number of times a (dis)entailment was detected by a constraint.

`prunings`
: The number of times a domain was pruned.

`backtracks`
: The number of times a contradiction was found by a domain being wiped out, or by a global constraint signalling failure. Other causes of backtracking, such as failed Prolog tests, are not covered by this counter.

`constraints`
: The number of constraints created.

`fd_statistics`
: Displays on the `user_error` stream a summary of the above statistics. All counters are zeroed.

30.6 Answer Constraints

By default, the answer constraint only shows the projection of the store onto the variables that occur in the query, but not any constraints that may be attached to these variables, nor any domains or constraints attached to other variables. This is a conscious decision, as no efficient algorithm for projecting answer constraints onto the query variables is known for this constraint system.

It is possible, however, to get a complete answer constraint including all variables that took part in the computation and their domains and attached constraints. This is done by asserting a clause for the following predicate:

`clpfd:full_answer`
: A dynamic hook predicate. If false (the default), the answer constraint only contains the domains of the query variables. If true, the answer constraints contains the domains and any attached constraints of all variables.

30.7 The Constraint System

30.7.1 Definitions

The constraint system is based on domain constraints and indexicals. A *domain constraint* is an expression $X::I$, where X is a domain variable and I is a nonempty set of integers.

A set S of domain constraints is called a *store*. *D(X,S)*, the *domain* of X in S, is defined as the intersection of all I such that $X::I$ belongs to S. The store is *contradictory* if the domain of some variable is empty; otherwise, it is *consistent*. A consistent store *S'* is an *extension* of a store S iff, for all variables X, *D(X,S')* is contained in *D(X,S)*.

The following definitions, adapted from [Van Hentenryck et al. 95], define important notions of consistency and entailment of constraints wrt. stores.

A ground constraint is *true* if it holds and *false* otherwise.

A constraint *C* is *domain-consistent wrt. S* iff, for each variable *Xi* and value *Vi* in *D(Xi,S)*, there exist values *Vj* in *D(Xj,S)*, *1 =< j =< n*, *i\=j*, such that *C(V1,...,Vn)* is true.

A constraint *C* is *domain-entailed by S* iff, for all values *Vj* in *D(Xj,S)*, *1 =< j =< n*, *C(V1,...,Vn)* is true.

Let *D'(X,S)* denote the interval *min(D(X,S))..max(D(X,S))*.

A constraint *C* is *interval-consistent wrt. S* iff, for each variable *Xi* there exist values *Vj* in *D'(Xj,S)*, *1 =< j =< n*, *i\=j*, such that *C(V1,...,min(D(Xi,S)),...,Vn)* and *C(V1,...,max(D(Xi,S)),...,Vn)* are both true.

A constraint *C* is *interval-entailed by S* iff, for all values *Vj* in *D'(Xj,S)*, *1 =< j =< n*, *C(V1,...,Vn)* is true.

Finally, a constraint is *domain-disentailed (interval-disentailed) by S* iff its negation is domain-entailed (interval-entailed) by *S*.

30.7.2 Pitfalls of Interval Reasoning

In most circumstances, arithmetic constraints only maintain interval-consistency and only detect interval-entailment and -disentailment. Note that there are cases where an interval-consistency maintaining constraint may detect a contradiction when the constraint is not yet interval-disentailed, as the following example illustrates. Note that *X* `#\=` *Y* maintains domain consistency if both arguments are constants or variables:

```
| ?- X+Y #= Z, X=1, Z=6, Y in 1..10, Y #\= 5.

no
| ?- X+Y #= Z #<=> B, X=1, Z=6, Y in 1..10, Y #\= 5.

X = 1,
Z = 6,
Y in(1..4)\/(6..10),
B in 0..1
```

Since `1+5#=6` holds, `X+Y #= Z` is not interval-disentailed, although any attempt to make it interval-consistent wrt. the store results in a contradictory store.

30.8 Defining Global Constraints

30.8.1 The Global Constraint Programming Interface

This section describes a programming interface by means of which new constraints can be written. The interface consists of a set of predicates provided by this library module. Constraints defined in this way can take arbitrary arguments and may use any constraint solving algorithm, provided it makes sense. Reification cannot be expressed in this interface; instead, reification may be achieved by explicitly passing a 0/1-variable to the constraint in question.

Global constraints have state which may be updated each time the constraint is resumed. The state information may be used e.g. in incremental constraint solving.

The following two predicates are the principal entrypoints for defining and posting new global constraints:

`clpfd:dispatch_global(+`*Constraint*`, +`*State0*`, -`*State*`, -`*Actions*`)`

A multifile hook predicate, telling the solver how to solve constraints of the form *Constraint*.

When defining a new constraint, a clause of this predicate must be added. Its body defines a constraint solving method and should always succeed deterministically. When a global constraint is called or resumed, the solver will call this predicate to deal with the constraint.

State0 and *State* are the old and new state respectively.

The constraint solving method must not invoke the constraint solver recursively e.g. by binding variables or posting new constraints; instead, *Actions* should be unified with a list of requests to the solver. Each request should be of the following form:

`exit` The constraint has become entailed, and ceases to exist.

`fail` The constraint has become disentailed, causing the solver to backtrack.

X `=` *V* The solver binds *X* to *V*.

X `in` *R* The solver constrains *X* to be a member of the *ConstantRange* *R* (see [Syntax of Indexicals], page 301).

X `in_set` *S*
The solver constrains *X* to be a member of the FD set *S* (see Section 30.8.3 [FD Set Operations], page 287).

`call(`*Goal*`)`
The solver calls the goal or constraint *Goal*, which should be module prefixed unless it is a built-in predicate or an exported predicate of the `clpfd` module.

`fd_global(+`*Constraint*`, +`*State*`, +`*Susp*`)`
: where *Constraint* is a constraint goal, *State* is its initial state, and *Susp* is a term encoding how the constraint should wake up in response to domain changes. This posts the constraint.

  Susp is a list of *F(Var)* terms where *Var* is a variable to suspend on and *F* is a functor encoding when to wake up:

  `dom(`*X*`)` wake up when the domain of *X* has changed

  `min(`*X*`)` wake up when the lower bound of *X* has changed

  `max(`*X*`)` wake up when the upper bound of *X* has changed

  `minmax(`*X*`)` wake up when the lower or upper of *X* has changed

  `val(`*X*`)` wake up when *X* has become ground

For an example of usage, see Section 30.8.4 [A Global Constraint Example], page 288.

30.8.2 Reflection Predicates

The constraint solving method needs access to information about the current domains of variables. This is provided by the following predicates, which are all constant time operations.

`fd_min(?`*X*`, ?`*Min*`)`
: where *X* is a domain variable (or an integer). *Min* is unified with the smallest value in the current domain of *X*, i.e. an integer or the atom `inf` denoting minus infinity.

`fd_max(?`*X*`, ?`*Max*`)`
: where *X* is a domain variable (or an integer). *Max* is unified with the upper bound of the current domain of *X*, i.e. an integer or the atom `sup` denoting infinity.

`fd_size(?`*X*`, ?`*Size*`)`
: where *X* is a domain variable (or an integer). *Size* is unified with the size of the current domain of *X*, if the domain is finite, or the atom `sup` otherwise.

`fd_set(?`*X*`, ?`*Set*`)`
: where *X* is a domain variable (or an integer). *Set* is unified with an FD set term denoting the internal representation of the current domain of *X*; see below.

`fd_dom(?`*X*`, ?`*Range*`)`
: where *X* is a domain variable (or an integer). *Range* is unified with a *ConstantRange* (see [Syntax of Indexicals], page 301) denoting the the current domain of *X*.

The following predicate can be used for computing the set of variables that are transitively connected via constraints to some given variables.

`fd_closure(+`*Vars*`, -`*Closure*`)`
: Given a list *Vars* of domain variables, *Closure* is the set of variables (including *Vars*) that can be transitively reached via constraints posted so far.

30.8.3 FD Set Operations

The domains of variables are internally represented compactly as *FD set* terms. The details of this representation are subject to change and should not be relied on. Therefore, a number of operations on FD sets are provided, as such terms play an important role in the interface. The following operations are the primitive ones:

`is_fdset(+`*Set*`)`
: *Set* is a valid FD set.

`empty_fdset(?`*Set*`)`
: *Set* is the empty FD set.

`fdset_parts(?`*Set*`, ?`*Min*`, ?`*Max*`, ?`*Rest*`)`
: *Set* is an FD set which is a union of the non-empty interval *Min*..*Max* and the FD set *Rest*, and all elements of *Rest* are greater than *Max*+1. *Min* and *Max* are both integers or the atoms `inf` and `sup`, denoting minus and plus infinity, respectively. Either *Set* or all the other arguments must be ground.

The following operations can all be defined in terms of the primitive ones, but in most cases, a more efficient implementation is used:

`empty_interval(+`*Min*`, +`*Max*`)`
: *Min*..*Max* is an empty interval.

`fdset_interval(?`*Set*`, ?`*Min*`, ?`*Max*`)`
: *Set* is an fdset which is the non-empty interval *Min*..*Max*.

`fdset_singleton(?`*Set*`, ?`*Elt*`)`
: *Set* is an FD set containing *Elt* only. At least one of the arguments must be ground.

`fdset_min(+`*Set*`, -`*Min*`)`
: *Min* is the lower bound of *Set*.

`fdset_max(+`*Set*`, -`*Min*`)`
: *Max* is the upper bound of *Set*. This operation is linear in the number of intervals of *Set*.

`fdset_size(+`*Set*`, -`*Size*`)`
: *Size* is the cardinality of *Set*, represented as `sup` if *Set* is infinite. This operation is linear in the number of intervals of *Set*.

`list_to_fdset(+`*List*`, -`*Set*`)`
: *Set* is the FD set containing the elements of *List*. Slightly more efficient if *List* is ordered.

`fdset_to_list(+`*Set*`, -`*List*`)`
: *List* is an ordered list of the elements of *Set*, which must be finite.

`range_to_fdset(+`*Range*`, -`*Set*`)`
: *Set* is the FD set containing the elements of the *ConstantRange* (see [Syntax of Indexicals], page 301) *Range*.

`fdset_to_range(+`*Set*`, -`*Range*`)`
: *Range* is a constant interval, a singleton constant set, or a union of such, denoting the same set as *Set*.

`fdset_add_element(+`*Set1*`, +`*Elt*` -`*Set2*`)`
: *Set2* is *Set1* with *Elt* inserted in it.

`fdset_del_element(+`*Set1*`, +`*Elt*`, -`*Set2*`)`
: *Set2* is like *Set1* but with *Elt* removed.

`fdset_disjoint(+`*Set1*`, +`*Set2*`)`
: The two FD sets have no elements in common.

`fdset_intersect(+`*Set1*`, +`*Set2*`)`
: The two FD sets have at least one element in common.

`fdset_intersection(+`*Set1*`, +`*Set2*`, -`*Intersection*`)`
: *Intersection* is the intersection between *Set1* and *Set2*.

`fdset_intersection(+`*Sets*`, -`*Intersection*`)`
: *Intersection* is the intersection of all the sets in *Sets*.

`fdset_member(?`*Elt*`, +`*Set*`)`
: is true when *Elt* is a member of *Set*. If *Elt* is unbound, *Set* must be finite.

`fdset_eq(+`*Set1*`, +`*Set2*`)`
: Is true when the two arguments represent the same set i.e. they are identical.

`fdset_subset(+`*Set1*`, +`*Set2*`)`
: Every element of *Set1* appears in *Set2*.

`fdset_subtract(+`*Set1*`, +`*Set2*`, -`*Difference*`)`
: *Difference* contains all and only the elements of *Set1* which are not also in *Set2*.

`fdset_union(+`*Set1*`, +`*Set2*`, -`*Union*`)`
: *Union* is the union of *Set1* and *Set2*.

`fdset_union(+`*Sets*`, -`*Union*`)`
: *Union* is the union of all the sets in *Sets*.

`fdset_complement(+`*Set*`, -`*Complement*`)`
: *Complement* is the complement of *Set* wrt. `inf..sup`.

30.8.4 A Global Constraint Example

The following example defines a new global constraint `exactly(`X`,`L`,`N`)` which is true if X occurs exactly N times in the list L of integers and domain variables. A version defined in terms of reified equalities was presented earlier; see Section 30.2.3 [Reified Constraints], page 274.

This example illustrates the use of state information. The state has two components: the list of variables that could still be X, and the number of variables still required to be X.

The constraint is defined to wake up on any domain change.

```
/*
An implementation of exactly(I, X[1]...X[m], N):

Necessary condition: 0 =< N =< m.
Rewrite rules:

[1] |= X[i]=I  ==> exactly(I, X[1]...X[i-1],X[i+1]...X[m], N-1):
[2] |= X[i]\=I ==> exactly(I, X[1]...X[i-1],X[i+1]...X[m], N):
[3] |= N=0     ==> X[1]\=I ... X[m]\=I
[4] |= N=m     ==> X[1]=I  ... X[m]=I
*/
:- use_module(library(clpfd)).

% the entrypoint
exactly(I, Xs, N) :-
        dom_suspensions(Xs, Susp),
        fd_global(exactly(I,Xs,N), state(Xs,N), Susp).

dom_suspensions([], []).
dom_suspensions([X|Xs], [dom(X)|Susp]) :-
        dom_suspensions(Xs, Susp).

% the solver method
:- multifile clpfd:dispatch_global/4.
clpfd:dispatch_global(exactly(I,_,_), state(Xs0,N0), state(Xs,N), Actions) :-
        exactly_solver(I, Xs0, Xs, N0, N, Actions).

exactly_solver(I, Xs0, Xs, N0, N, Actions) :-
        ex_filter(Xs0, Xs, N0, N, I),
        length(Xs, M),
        (   N=:=0 -> Actions = [exit|Ps], ex_neq(Xs, I, Ps)
        ;   N=:=M -> Actions = [exit|Ps], ex_eq(Xs, I, Ps)
        ;   N>0, N<M -> Actions = []
        ;   Actions = [fail]
        ).

% rules [1,2]: filter the X's, decrementing N
ex_filter([], [], N, N, _).
ex_filter([X|Xs], Ys, L, N, I) :- X==I, !,
        M is L-1,
        ex_filter(Xs, Ys, M, N, I).
ex_filter([X|Xs], Ys0, L, N, I) :-
        fd_set(X, Set),
        fdset_member(I, Set), !,
        Ys0 = [X|Ys],
        ex_filter(Xs, Ys, L, N, I).
ex_filter([_|Xs], Ys, L, N, I) :-
        ex_filter(Xs, Ys, L, N, I).
```

```
% rule [3]: all must be neq I
ex_neq(Xs, I, Ps) :-
        fdset_singleton(Set0, I),
        fdset_complement(Set0, Set),
        eq_all(Xs, Set, Ps).

% rule [4]: all must be eq I
ex_eq(Xs, I, Ps) :-
        fdset_singleton(Set, I),
        eq_all(Xs, Set, Ps).

eq_all([], _, []).
eq_all([X|Xs], Set, [X in_set Set|Ps]) :-
        eq_all(Xs, Set, Ps).

end_of_file.

% sample queries:
| ?- exactly(5,[A,B,C],1), A=5.

A = 5,
B in(inf..4)\/(6..sup),
C in(inf..4)\/(6..sup)

| ?- exactly(5,[A,B,C],1), A in 1..2, B in 3..4.

C = 5,
A in 1..2,
B in 3..4
```

30.9 Defining Primitive Constraints

Indexicals are the principal means of defining constraints, but it is usually not necessary to resort to this level of programming—most commonly used constraints are available in a library and/or via macro-expansion. The key feature about indexicals is that they give the programmer precise control over aspects of the operational semantics of the constraints. Trade-offs can be made between the computational cost of the constraints and their pruning power. The indexical language provides many degrees of freedom for the user to select the level of consistency to be maintained depending on application-specific needs.

30.9.1 Indexicals

An *indexical* is a reactive functional rule of the form *X* `in` *R*, where *R* is a set valued *range expression* (see below). See [Syntax of Indexicals], page 301, for a grammar defining indexicals and range expressions.

Indexicals can play one of two roles: *propagating indexicals* are used for constraint solving, and *checking indexicals* are used for entailment checking. When a propagating indexical fires, *R* is evaluated in the current store *S*, which is extended by adding the new domain constraint *X*::*S(R)* to the store, where *S(R)* denotes the value of *R* in *S*. When a checking indexical fires, it checks if *D(X,S)* is contained in *S(R)*, and if so, the constraint corresponding to the indexical is detected as entailed.

30.9.2 Range Expressions

A range expression has one of the following forms, where *Ri* denote range expressions, *Ti* denote integer valued *term expressions*, *S(Ti)* denotes the integer value of *Ti* in *S*, *X* denotes a variable, *I* denotes an integer, and *S* denotes the current store.

`dom(`*X*`)` evaluates to *D(X,S)*

`{`*T1*`,...,`*Tn*`}`
: evaluates to `{S(`*T1*`),...,S(`*Tn*`)}`. Any term expression containing a subexpression which is a variable that is not "quantified" by `unionof/3` will only be evaluated when this variable has been assigned.

T1`..`*T2* evaluates to the interval between *S(T1)* and *S(T2)*.

R1`/\`*R2* evaluates to the intersection of *S(R1)* and *S(R2)*

R1`\/`*R2* evaluates to the union of *S(R1)* and *S(R2)*

`\`*R2* evaluates to the complement of *S(R2)*

R1`+`*R2*
R1`+`*T2* evaluates to *S(R2)* or *S(T2)* added pointwise to *S(R1)*

`-`*R2* evaluates to *S(R2)* negated pointwise

R1`-`*R2*
R1`-`*T2*
T1`-`*R2* evaluates to *S(R2)* or *S(T2)* subtracted pointwise from *S(R1)* or *S(T1)*

R1 `mod` *R2*
R1 `mod` *T2*
: evaluates to *S(R1)* pointwise modulo *S(R2)* or *S(T2)*

R1 `?` *R2* evaluates to *S(R2)* if *S(R1)* is a non-empty set; otherwise, evaluates to the empty set. This expression is commonly used in the context `(`*R1* `? (inf..sup) \/` *R3*`)`, which evaluates to *S(R3)* if *S(R1)* is an empty set; otherwise, evaluates to `inf..sup`. As an optimization, *R3* is not evaluated while the value of *R1* is a non-empty set.

`unionof(`*X*`,`*R1*`,`*R2*`)`
: evaluates to the union of *S(Expr_1)...S(Expr_N)*, where each *Expr_I* has been formed by substituting *K* for *X* in *R2*, where *K* is the *I*:th element of *S(R1)*. See Section 30.10.2 [N Queens], page 298, for an example of usage. **N.B.** If *S(R1)* is infinite, the evaluation of the indexical will be abandoned, and the indexical will simply suspend.

`switch(`*T1*`,`*MapList*`)`
: evaluates to *S(Expr)* if *S(T1)* equals *Key* and *MapList* contains a pair *Key*`-`*Expr*. Otherwise, evaluates to the empty set.

When used in the body of an FD predicate (see Section 30.9.8 [Goal Expanded Constraints], page 297), a `relation/3` expression expands to two indexicals, each consisting of a `switch/2` expression nested inside a `unionof/3` expression. Thus, the following constraints are equivalent:

```
p(X, Y) +: relation(X, [1-{1},2-{1,2},3-{1,2,3}], Y).

q(X, Y) +:
        X in unionof(B,dom(Y),switch(B,[1-{1,2,3},2-{2,3},3-{3}])),
        Y in unionof(B,dom(X),switch(B,[1-{1},2-{1,2},3-{1,2,3}])).
```

30.9.3 Term Expressions

A term expression has one of the following forms, where *T1* and *T2* denote term expressions, *X* denotes a variable, *I* denotes an integer, and *S* denotes the current store.

`min(`*X*`)` evaluates to the minimum of *D(X,S)*

`max(`*X*`)` evaluates to the maximum of *D(X,S)*

`card(`*X*`)` evaluates to the size of *D(X,S)*

X evaluates to the integer value of *X*. A subexpression of this form, not "quantified" by `unionof/3`, will cause the evaluation to suspend until the variable is assigned.

I an integer

`inf` minus infinity

`sup` plus infinity

`-`*T1* evaluates to *S(T1)* negated

T1`+`*T2* evaluates to the sum of *S(T1)* and *S(T2)*

T1`-`*T2* evaluates to the difference of *S(T1)* and *S(T2)*

T1`*`*T2* evaluates to the product of *S(T1)* and *S(T2)*, where *S(T2)* must not be negative

T1`/>`*T2* evaluates to the quotient of *S(T1)* and *S(T2)*, rounded up, where *S(T2)* must be positive

T1`/<`*T2* evaluates to the quotient of *S(T1)* and *S(T2)*, rounded down, where *S(T2)* must be positive

T1 `mod` *T2*
: evaluates to the modulo of *S(T1)* and *S(T2)*

30.9.4 Monotonicity of Indexicals

A range R is *monotone in* S iff the value of R in S' is contained in the value of R in S, for every extension S' of S. A range R is *anti-monotone in* S iff the value of R in S is contained in the value of R in S', for every extension S' of S. By abuse of notation, we will say that X `in` R is (anti-)monotone iff R is (anti-)monotone.

The consistency or entailment of a constraint C expressed as indexicals X `in` R in a store S is checked by considering the relationship between *D(X,S)* and *S(R)*, together with the (anti-)monotonicity of R in S. The details are given in Section 30.9.6 [Execution of Propagating Indexicals], page 296 and Section 30.9.7 [Execution of Checking Indexicals], page 296.

The solver checks (anti-)monotonicity by requiring that certain variables occurring in the indexical be ground. This sufficient condition can sometimes be false for an (anti-)monotone indexical, but such situations are rare in practice.

30.9.5 FD predicates

The following example defines the constraint X+Y=T as an *FD predicate* in terms of three indexicals. Each indexical is a rule responsible for removing values detected as incompatible from one particular constraint argument. Indexicals are *not* Prolog goals; thus, the example does not express a conjunction. However, an indexical may make the store contradictory, in which case backtracking is triggered:

```
plus(X,Y,T) +:
        X in min(T) - max(Y) .. max(T) - min(Y),
        Y in min(T) - max(X) .. max(T) - min(X),
        T in min(X) + min(Y) .. max(X) + max(Y).
```

The above definition contains a single clause used for constraint solving. The first indexical wakes up whenever the bounds of *S(T)* or *S(Y)* are updated, and removes from *D(X,S)* any values that are not compatible with the new bounds of T and Y. Note that in the event of "holes" in the domains of T or Y, *D(X,S)* may contain some values that are incompatible with X+Y=T but go undetected. Like most built-in arithmetic constraints, the above definition maintains interval-consistency, which is significantly cheaper to maintain than domain-consistency and suffices in most cases. The constraint could for example be used as follows:

```
| ?- X in 1..5, Y in 2..8, plus(X,Y,T).

X in 1..5,
Y in 2..8,
T in 3..13 ?

yes
```

Thus, when an FD predicate is called, the '`+:`' clause is activated.

The definition of a user constraint has to specify what domain constraints should be added to the constraint store when the constraint is posted. Therefore the FD predicate contains a set of indexicals, each representing a domain constraint to be added to the constraint store. The actual domain constraint depends on the constraint store itself. For example, the third indexical in the above FD predicate prescribes the domain constraint '`T :: 3..13`' if the store contains '`X :: 1..5, Y :: 2..8`'. As the domain of some variables gets narrower, the indexical may enforce a new, stricter constraint on some other variables. Therefore such an indexical (called a propagating indexical) can be viewed as an agent reacting to the changes in the store by enforcing further changes in the store.

In general there are three stages in the lifetime of a propagating indexical. When it is posted it may not be evaluated immediately (e.g. has to wait until some variables are ground before being able to modify the store). Until the preconditions for the evaluation are satisfied, the agent does not enforce any constraints. When the indexical becomes evaluable the resulting domain constraint is added to the store. The agent then waits and reacts to changes in the domains of variables occurring in the indexical by re-evaluating it and adding the new, stricter constraint to the store. Eventually the computation reaches a phase when no further refinement of the store can result in a more precise constraint (the indexical is entailed by the store), and then the agent can cease to exist.

A necessary condition for the FD predicate to be correctly defined is the following: for any store mapping each variable to a singleton domain the execution of the indexicals should succeed without contradiction exactly when the predicate is intended to be true.

There can be several alternative definitions for the same user constraint with different strengths in propagation. For example, the definition of `plusd` below encodes the same `X+Y=T` constraint as the `plus` predicate above, but maintaining domain consistency:

```
plusd(X,Y,T) +:
        X in dom(T) - dom(Y),
        Y in dom(T) - dom(X),
        T in dom(X) + dom(Y).

| ?- X in {1}\/{3}, Y in {10}\/{20}, plusd(X, Y, T).

X in{1}\/{3},
Y in{10}\/{20},
T in{11}\/{13}\/{21}\/{23} ?

yes
```

This costs more in terms of execution time, but gives more precise results. For singleton domains `plus` and `plusd` behave in the same way.

In our design, general indexicals can only appear in the context of FD predicate definitions. The rationale for this restriction is the need for general indexicals to be able to suspend and resume, and this ability is only provided by the FD predicate mechanism.

If the program merely posts a constraint, it suffices for the definition to contain a single clause for solving the constraint. If a constraint is reified or occurs in a propositional formula, the definition must contain four clauses for solving and checking entailment of the constraint and its negation. The role of each clause is reflected in the "neck" operator. The following table summarizes the different forms of indexical clauses corresponding to a constraint C. In all cases, *Head* should be a compound term with all arguments being distinct variables:

Head `+:` *Indexicals*.
: The clause consists of propagating indexicals for solving C.

Head `-:` *Indexicals*.
: The clause consists of propagating indexicals for solving the negation of C.

Head `+?` *Indexical*.
: The clause consists of a single checking indexical for testing entailment of C.

Head `-?` *Indexical*.
: The clause consists of a single checking indexical for testing entailment of the negation of C.

When a constraint is reified, the solver spawns two reactive agents corresponding to detecting entailment and disentailment. Eventually, one of them will succeed in this and consequently will bind B to 0 or 1. A third agent is spawned, waiting for B to become assigned, at which time the constraint (or its negation) is posted. In the mean time, the constraint may have been detected as (dis)entailed, in which case the third agent is dismissed. The waiting is implemented by means of the coroutining facilities of SICStus Prolog.

As an example of a constraint with all methods defined, consider the following library constraint defining a disequation between two domain variables:

```
'x\\=y'(X,Y) +:
        X in \{Y},
        Y in \{X}.
'x\\=y'(X,Y) -:
        X in dom(Y),
        Y in dom(X).
'x\\=y'(X,Y) +?
        X in \dom(Y).
'x\\=y'(X,Y) -?
        X in {Y}.
```

The following sections provide more precise coding rules and operational details for indexicals. X `in` R denotes an indexical corresponding to a constraint C. S denotes the current store.

30.9.6 Execution of Propagating Indexicals

Consider the definition of a constraint C containing a propagating indexical X `in` R. Let *TV(X,C,S)* denote the set of values for X that can make C true in some ground extension of the store S. Then the indexical should obey the following coding rules:

- all arguments of C except X should occur in R
- if R is ground in S, *S(R) = TV(X,C,S)*

If the coding rules are observed, *S(R)* can be proven to contain *TV(X,C,S)* for all stores in which R is monotone. Hence it is natural for the implementation to wait until R becomes monotone before admitting the propagating indexical for execution. The execution of X `in` R thus involves the following:

- If *D(X,S)* is disjoint from *S(R)*, a contradiction is detected.
- If *D(X,S)* is contained in *S(R)*, *D(X,S)* does not contain any values known to be incompatible with C, and the indexical suspends, unless R is ground in S, in which case C is detected as entailed.
- Otherwise, *D(X,S)* contains some values that are known to be incompatible with C. Hence, $X::$*S(R)* is added to the store (X is *pruned*), and the indexical suspends, unless R is ground in S, in which case C is detected as entailed.

A propagating indexical is scheduled for execution as follows:

- it is evaluated initially as soon as it has become monotone
- it is re-evaluated when one of the following conditions occurs:
  1. the domain of a variable Y that occurs as `dom(`Y`)` or `card(`Y`)` in R has been updated
  2. the lower bound of a variable Y that occurs as `min(`Y`)` in R has been updated
  3. the upper bound of a variable Y that occurs as `max(`Y`)` in R has been updated

30.9.7 Execution of Checking Indexicals

Consider the definition of a constraint C containing a checking indexical X `in` R. Let *FV(X,C,S)* denote the set of values for X that can make C false in some ground extension of the store S. Then the indexical should obey the following coding rules:

- all arguments of C except X should occur in R
- if R is ground in S, *S(R) = TV(X,C,S)*

If the coding rules are observed, *S(R)* can be proven to exclude *FV(X,C,S)* for all stores in which R is anti-monotone. Hence it is natural for the implementation to wait until R becomes anti-monotone before admitting the checking indexical for execution. The execution of X `in` R thus involves the following:

- If *D(X,S)* is contained in *S(R)*, none of the possible values for X can make C false, and so C is detected as entailed.

- Otherwise, if *D(X,S)* is disjoint from *S(R)* and R is ground in S, all possible values for X will make C false, and so C is detected as disentailed.
- Otherwise, *D(X,S)* contains some values that could make C true and some that could make C false, and the indexical suspends.

A checking indexical is scheduled for execution as follows:

- it is evaluated initially as soon as it has become anti-monotone
- it is re-evaluated when one of the following conditions occurs:
  1. the domain of X has been pruned, or X has been assigned
  2. the domain of a variable Y that occurs as `dom(`Y`)` or `card(`Y`)` in R has been pruned
  3. the lower bound of a variable Y that occurs as `min(`Y`)` in R has been increased
  4. the upper bound of a variable Y that occurs as `max(`Y`)` in R has been decreased

30.9.8 Goal Expanded Constraints

The arithmetic, membership, and propositional constraints described earlier are transformed at compile time to conjunctions of goals of library constraints.

Sometimes it is necessary to postpone the expansion of a constraint until runtime, e.g. if the arguments are not instantiated enough. This can be achieved by wrapping `call/1` around the constraint.

Although space economic (linear in the size of the source code), the expansion of a constraint to library goals can have an overhead compared to expressing the constraint in terms of indexicals. Temporary variables holding intermediate values may have to be introduced, and the grain size of the constraint solver invocations can be rather small. The translation of constraints to library goals has been greatly improved in the current version, so these problems have virtually disappeared. However, for backward compatibility, an implementation by compilation to indexicals of the same constraints is also provided. An FD predicate may be defined by a single clause:

Head `+:` *Constraint*.

where *Constraint* is an arithmetic constraint or an `element/3` or a `relation/3` constraint. This translation is only available for '`+:`' clauses; thus, *Head* cannot be reified.

In the case of arithmetic constraints, the constraint must be over linear terms (see [Syntax of Indexicals], page 301). The memory consumption of the FD predicate will be quadratic in the size of the source code. The alternative version of `sum/8` in Section 30.10.1 [Send More Money], page 298 illustrates this technique.

In the case of `element(X,L,Y)` or `relation(X,L,Y)`, the memory consumption of the FD predicate will be linear in the size of the source code. The execution time of the initial evaluation of the FD predicate will be linear in the size of the initial domains for X and Y; if these domains are infinite, no propagation will take place.

30.10 Example Programs

This section contains a few example programs. The first two programs are included in a benchmark suite that comes with the distribution. The benchmark suite is run by typing:

```
| ?- compile(library('clpfd/examples/bench')).
| ?- bench.
```

30.10.1 Send More Money

Let us return briefly to the Send More Money problem (see Section 30.2.2 [A Constraint Satisfaction Problem], page 273). Its `sum/8` predicate will expand to a space-efficient conjunction of library constraints. A faster but more memory consuming version is defined simply by changing the neck symbol of `sum/8` from '`:-`' to '`+:`', thus turning it into an FD predicate:

```
sum(S, E, N, D, M, O, R, Y) +:
                   1000*S + 100*E + 10*N + D
     +             1000*M + 100*O + 10*R + E
     #= 10000*M + 1000*O + 100*N + 10*E + Y.
```

30.10.2 N Queens

The problem is to place N queens on an NxN chess board so that no queen is threatened by another queen.

The variables of this problem are the N queens. Each queen has a designated row. The problem is to select a column for it.

The main constraint of this problem is that no queen threaten another. This is encoded by the `no_threat/3` constraint and holds between all pairs `(X,Y)` of queens. It could be defined as

```
no_threat(X, Y, I) :-
        X   #\= Y,
        X+I #\= Y,
        X-I #\= Y.
```

However, this formulation introduces new temporary domain variables and creates twelve fine-grained indexicals. Worse, the arithmetic constraints are only guaranteed to maintain interval-consistency and so may miss some opportunities for pruning elements in the middle of domains.

A better idea is to formulate `no_threat/3` as an FD predicate with two indexicals, as shown in the program below. This constraint will not fire until one of the queens has been assigned (the corresponding indexical does not become monotone until then). Hence, the constraint is still not as strong as it could be.

For example, if the domain of one queen is (2..3), then it will threaten any queen placed in column 2 or 3 on an adjacent row, no matter which of the two open positions is chosen for the first queen.

The commented out formulation of the constraint captures this reasoning, and illustrates the use of the `unionof/3` operator. This stronger version of the constraint indeed gives less backtracking, but is computationally more expensive and does not pay off in terms of execution time, except possibly for very large chess boards.

It is clear that `no_threat/3` cannot detect any incompatible values for a queen with domain of size greater than three. This observation is exploited in the third version of the constraint.

The first-fail principle is appropriate in the enumeration part of this problem.

```
:- use_module(library(clpfd)).

queens(N, L, LabelingType) :-
     length(L, N),
     domain(L, 1, N),
     constrain_all(L),
     labeling(LabelingType, L).

constrain_all([]).
constrain_all([X|Xs]) :-
     constrain_between(X, Xs, 1),
     constrain_all(Xs).

constrain_between(_X, [], _N).
constrain_between(X, [Y|Ys], N) :-
     no_threat(X, Y, N),
     N1 is N+1,
     constrain_between(X, Ys, N1).


% version 1: weak but efficient
no_threat(X, Y, I) +:
     X in \({Y} \/ {Y+I} \/ {Y-I}),
     Y in \({X} \/ {X+I} \/ {X-I}).

/*
% version 2: strong but very inefficient version
no_threat(X, Y, I) +:
    X in unionof(B,dom(Y),\({B} \/ {B+I} \/ {B-I})),
    Y in unionof(B,dom(X),\({B} \/ {B+I} \/ {B-I})).

% version 3: strong but somewhat inefficient version
no_threat(X, Y, I) +:
    X in (4..card(Y)) ? (inf..sup) \/
          unionof(B,dom(Y),\({B} \/ {B+I} \/ {B-I})),
    Y in (4..card(X)) ? (inf..sup) \/
          unionof(B,dom(X),\({B} \/ {B+I} \/ {B-I})).
*/
```

```
| ?- queens(8, L, [ff]).

L = [1,5,8,6,3,7,2,4] ?
```

30.10.3 Cumulative Scheduling

This example is a very small scheduling problem. We consider seven tasks where each task has a fixed duration and a fixed amount of used resource:

```
TASK    DURATION        RESOURCE
====    ========        ========
t1            16               2
t2             6               9
t3            13               3
t4             7               7
t5             5              10
t6            18               1
t7             4              11
```

The goal is to find a schedule that minimizes the completion time for the schedule while not exceeding the capacity 13 of the resource. The resource constraint is succinctly captured by a `cumulative/4` constraint. Branch-and-bound search is used to find the minimal completion time.

This example was borrowed from [Beldiceanu & Contejean 94].

```
:- use_module(library(clpfd)).

schedule(Ss, End) :-
        length(Ss, 7),
        Ds = [16, 6,13, 7, 5,18, 4],
        Rs = [ 2, 9, 3, 7,10, 1,11],
        domain(Ss, 1, 30),
        domain([End], 1, 50),
        after(Ss, Ds, End),
        cumulative(Ss, Ds, Rs, 13),
        labeling([minimize(End)], [End|Ss]).

after([], [], _).
after([S|Ss], [D|Ds], E) :- E #>= S+D, after(Ss, Ds, E).

%% End of file

| ?- schedule(Ss, End).

Ss = [1,17,10,10,5,5,1],
End = 23 ?
```

30.11 Syntax Summary

Syntax of Indexicals

```
X --> variable                  { domain variable }

Constant --> integer
  |   inf                       { minus infinity }
  |   sup                       { plus infinity }

Term --> Constant
  |   X                         { suspend until assigned }
  |   min(X)                    { min. of domain of X }
  |   max(X)                    { max. of domain of X }
  |   card(X)                   { size of domain of X }
  |   - Term
  |   Term + Term
  |   Term - Term
  |   Term * Term
  |   Term /> Term               { division rounded up }
  |   Term /< Term               { division rounded down }
  |   Term mod Term

TermSet --> {Term,...,Term}

Range --> TermSet
  |   dom(X)                     { domain of X }
  |   Term..Term                 { interval }
  |   Range/\Range               { intersection }
  |   Range\/Range               { union }
  |   \Range                     { complement }
  |   - Range                    { pointwise negation }
  |   Range + Range              { pointwise addition }
  |   Range - Range              { pointwise subtraction }
  |   Range mod Range            { pointwise modulo }
  |   Range + Term               { pointwise addition }
  |   Range - Term               { pointwise subtraction }
  |   Term - Range               { pointwise subtraction }
  |   Range mod Term             { pointwise modulo }
  |   Range ? Range
  |   unionof(X,Range,Range)
  |   switch(Term,MapList)

ConstantSet --> {integer,...,integer}

ConstantRange --> ConstantSet
  |   Constant..Constant
  |   ConstantRange/\ConstantRange
  |   ConstantRange\/ConstantRange
  |   \ConstantRange

MapList --> []
  |   [integer-ConstantRange|MapList]

CList --> []
  |   [integer|CList]
```

Indexical `-->` *X* `in` *Range*

Indexicals `-->` *Indexical*
`|` *Indexical*`,` *Indexicals*

ConstraintBody `-->` *Indexicals*
`|` *LinExpr RelOp LinExpr*
`|` `element(`*X*`,`*CList*`,`*X*`)`
`|` `relation(`*X*`,`*MapList*`,`*X*`)`

Head `-->` *term* `{` a compound term with unique variable args `}`

TellPos `-->` *Head* `+:` *ConstraintBody*`.`
TellNeg `-->` *Head* `-:` *ConstraintBody*`.`
AskPos `-->` *Head* `+?` *Indexical*`.`
AskNeg `-->` *Head* `-?` *Indexical*`.`

ConstraintDef `-->`
TellPos`.` `[`*TellNeg*`.]` `[`*AskPos*`.]` `[`*AskNeg*`.]`

Syntax of Arithmetic Expressions

X `-->` *variable* `{` domain variable `}`

N `-->` *integer*

LinExpr `-->` *N* `{` linear expression `}`
`|` *X*
`|` *N* `*` *X*
`|` *N* `*` *N*
`|` *LinExpr* `+` *LinExpr*
`|` *LinExpr* `-` *LinExpr*

Expr `-->` *LinExpr*
`|` *Expr* `+` *Expr*
`|` *Expr* `-` *Expr*
`|` *Expr* `*` *Expr*
`|` *Expr* `/` *Expr* `{` integer division `}`
`|` *Expr* `mod` *Expr*
`|` `min(`*Expr*`,`*Expr*`)`
`|` `max(`*Expr*`,`*Expr*`)`
`|` `abs(`*Expr*`)`

RelOp `-->` `#=` | `#\=` | `#<` | `#=<` | `#>` | `#>=`

Operator Declarations

```
:- op(1200, xfx, [+:,-:,+?,-?]).
:- op(760,  yfx, #<=>).
:- op(750,  xfy, #=>).
:- op(750,  yfx, #<=).
:- op(740,  yfx, #\/).
:- op(730,  yfx, #\).
:- op(720,  yfx, #/\).
:- op(710,   fy, #\).
:- op(700,  xfx, [in,in_set]).
:- op(700,  xfx, [#=,#\=,#<,#=<,#>,#>=]).
:- op(550,  xfx, ..).
:- op(500,   fy, \).
:- op(490,  yfx, ?).
:- op(400,  yfx, [/>,/<]).
```

31 Constraint Handling Rules

Copyright

This chapter is Copyright © 1996-98 LMU

LMU (Ludwig-Maximilians-University)
Munich, Germany

Permission is granted to make and distribute verbatim copies of this chapter provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this chapter under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this chapter into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by LMU.

31.1 Introduction

Experience from real-life applications using constraint-based programming has shown that typically, one is confronted with a heterogeneous mix of different types of constraints. To be able to express constraints as they appear in the application and to write and combine constraint systems, a special purpose language for writing constraint systems called *constraint handling rules* (CHR) was developed. CHR have been used to encode a wide range of constraint handlers (solvers), including new domains such as terminological and temporal reasoning. Several CHR libraries exist in declarative languages such as Prolog and LISP, worldwide more than 20 projects use CHR. You can find more information about CHR at URL: `http://www.pst.informatik.uni-muenchen.de/personen/fruehwir/chr-intro.html`

The high-level CHR are an excellent tool for rapid prototyping and implementation of constraint handlers. The usual abstract formalism to describe a constraint system, i.e. inference rules, rewrite rules, sequents, formulas expressing axioms and theorems, can be written as CHR in a straightforward way. Starting from this executable specification, the rules can be refined and adapted to the specifics of the application.

The CHR library includes a compiler, which translates CHR programs into Prolog programs on the fly, and a runtime system, which includes a stepper for debugging. Many constraint handlers are provided in the example directory of the library.

CHR are essentially a committed-choice language consisting of guarded rules that rewrite constraints into simpler ones until they are solved. CHR define both *simplification* of and *propagation*

over constraints. Simplification replaces constraints by simpler constraints while preserving logical equivalence (e.g. `X>Y,Y>X <=> fail`). Propagation adds new constraints which are logically redundant but may cause further simplification (e.g. `X>Y,Y>Z ==> X>Z`). Repeatedly applying CHR incrementally simplifies and finally solves constraints (e.g. `A>B,B>C,C>A` leads to `fail`.

With multiple heads and propagation rules, CHR provide two features which are essential for non-trivial constraint handling. The declarative reading of CHR as formulas of first order logic allows one to reason about their correctness. On the other hand, regarding CHR as a rewrite system on logical formulas allows one to reason about their termination and confluence.

In case the implementation of CHR disagrees with your expectations based on this chapter, drop a line to the current maintainer: `christian@ai.univie.ac.at` (Christian Holzbaur).

31.2 Introductory Examples

We define a CHR constraint for less-than-or-equal, `leq`, that can handle variable arguments. This handler can be found in the library as the file `leq.pl`. (The code works regardless of options switched on or off.)

```
:- use_module(library(chr)).

handler leq.
constraints leq/2.
:- op(500, xfx, leq).

reflexivity  @ X leq Y <=> X=Y | true.
antisymmetry @ X leq Y , Y leq X <=> X=Y.
idempotence  @ X leq Y \ X leq Y <=> true.
transitivity @ X leq Y , Y leq Z ==> X leq Z.
```

The CHR specify how `leq` simplifies and propagates as a constraint. They implement reflexivity, idempotence, antisymmetry and transitivity in a straightforward way. CHR `reflexivity` states that `X leq Y` simplifies to `true`, provided it is the case that `X=Y`. This test forms the (optional) guard of a rule, a precondition on the applicability of the rule. Hence, whenever we see a constraint of the form `A leq A` we can simplify it to `true`.

The rule `antisymmetry` means that if we find `X leq Y` as well as `Y leq X` in the constraint store, we can replace it by the logically equivalent `X=Y`. Note the different use of `X=Y` in the two rules: In the `reflexivity` rule the equality is a precondition (test) on the rule, while in the `antisymmetry` rule it is enforced when the rule fires. (The reflexivity rule could also have been written as `reflexivity X leq X <=> true`.)

The rules `reflexivity` and `antisymmetry` are *simplification CHR*. In such rules, the constraints found are removed when the rule applies and fires. The rule `idempotence` is a *simpagation CHR*, only the constraints right of `'\'` will be removed. The rule says that if we find `X leq Y` and another `X leq Y` in the constraint store, we can remove one.

Finally, the rule `transitivity` states that the conjunction `X leq Y, Y leq Z` implies `X leq Z`. Operationally, we add `X leq Z` as (redundant) constraint, without removing the constraints `X leq Y, Y leq Z`. This kind of CHR is called *propagation CHR*.

Propagation CHR are useful, as the query `A leq B,C leq A,B leq C` illustrates: The first two constraints cause CHR `transitivity` to fire and add `C leq B` to the query. This new constraint together with `B leq C` matches the head of CHR `antisymmetry`, `X leq Y, Y leq X`. So the two constraints are replaced by `B=C`. Since `B=C` makes `B` and `C` equivalent, CHR `antisymmetry` applies to the constraints `A leq B, C leq A`, resulting in `A=B`. The query contains no more CHR constraints, the simplification stops. The constraint handler we built has solved `A leq B, C leq A, B leq C` and produced the answer `A=B, B=C`:

```
A leq B,C leq A,B leq C.
% C leq A, A leq B propagates C leq B by transitivity.
% C leq B, B leq C simplifies to B=C by antisymmetry.
% A leq B, C leq A simplifies to A=B by antisymmetry since B=C.
A=B,B=C.
```

Note that multiple heads of rules are essential in solving these constraints. Also note that this handler implements a (partial) order constraint over any constraint domain, this generality is only possible with CHR.

As another example, we can implement the sieve of Eratosthenes to compute primes simply as (for variations see the handler '`primes.pl`'):

```
:- use_module(library(chr)).
handler eratosthenes.
constraints primes/1,prime/1.

primes(1) <=> true.
primes(N) <=> N>1 | M is N-1,prime(N),primes(M). % generate candidates

absorb(J) @ prime(I) \ prime(J) <=> J mod I =:= 0 | true.
```

The constraint `primes(N)` generates candidates for prime numbers, `prime(M)`, where `M` is between `1` and `N`. The candidates react with each other such that each number absorbs multiples of itself. In the end, only prime numbers remain.

Looking at the two rules defining `primes/1`, note that head matching is used in CHR, so the first rule will only apply to `primes(1)`. The test `N>1` is a guard (precondition) on the second rule. A call with a free variable, like `primes(X)`, will delay (suspend). The third, multi-headed rule `absorb(J)` reads as follows: If there is a constraint `prime(I)` and some other constraint `prime(J)` such that `J mod I =:= 0` holds, i.e. `J` is a multiple of `I`, then keep `prime(I)` but remove `prime(J)` and execute the body of the rule, `true`.

31.3 CHR Library

CHR extend the Prolog syntax by a few constructs introduced in the next sections. Technically, the extension is achieved through the `user:term_expansion/2` mechanism. A file that contains a constraint handler may also contain arbitrary Prolog code. Constraint handling rules can be scattered across a file. Declarations and options should precede rules. There can only be at most one constraint handler per module.

31.3.1 Loading the Library

Before you can load or compile any file containing a constraint handler (solver) written in CHR, the `chr` library module has to be imported:

```
| ?- use_module(library(chr)).
```

It is recommended to include the corresponding directive at the start of your files containing handlers:

```
:- use_module(library(chr)).
```

31.3.2 Declarations

Declarations in files containing CHR affect the compilation and thus the behavior of the rules at runtime.

The mandatory handler declaration precedes any other CHR specific code. Example:

```
handler minmax.
```

A handler name must be a valid Prolog `atom`. Per module, only one constraint handler can be defined.

The constraints must be declared before they are used by rules. With this mandatory declaration one lists the constraints the rules will later talk about. The declaration can be used more than once per handler. Example:

```
constraints  leq/2, minimum/3, maximum/3.
```

The following optional declaration allows for conditional rule compilation. Only the rules mentioned get compiled. Rules are referred to by their names (see Section 31.3.3 [CHR Syntax], page 309). The latest occurrence takes precedence if used more than once per handler. Although it can be put anywhere in the handler file, it makes sense, as with other declarations, to use it early. Example:

```
rules antisymmetry, transitivity.
```

To simplify the handling of operator declarations, in particular during `fcompile/1`, `operator/3` declarations with the same denotation as `op/3`, but taking effect during compilation and loading, are helpful. Example:

```
operator(700, xfx, ::).
operator(600, xfx, :).
```

31.3.3 Constraint Handling Rules, Syntax

A constraint handling rule has one or more heads, an optional guard, a body and an optional name. A *Head* is a *Constraint*. A constraint is a callable Prolog term, whose functor is a declared constraint. The *Guard* is a Prolog goal. The *Body* of a rule is a Prolog goal (including constraints). A rule can be named with a *Name* which can be any Prolog term (including variables from the rule).

There are three kinds of constraint handling rules:

```
Rule           --> [Name @]
                     (Simplification | Propagation | Simpagation)
                        [pragma Pragma].

Simplification --> Heads         <=> [Guard '|'] Body
Propagation    --> Heads         ==> [Guard '|'] Body
Simpagation    --> Heads \ Heads <=> [Guard '|'] Body

Heads          --> Head | Head, Heads
Head           --> Constraint | Constraint # Id
Constraint     --> a callable term declared as constraint
Id             --> a unique variable

Guard          --> Ask | Ask & Tell
Ask            --> Goal
Tell           --> Goal
Goal           --> a callable term, including conjunction and disjunction etc.

Body           --> Goal

Pragma         --> a conjunction of terms usually referring to
                   one or more heads identified via #/2
```

The symbol '`|`' separates the guard (if present) from the body of a rule. Since '`|`' is read as '`;`' (disjunction) by the reader, care has to be taken when using disjunction in the guard or body of the rule. The top level disjunction will always be interpreted as guard-body separator '`|`', so proper bracketing has to be used, e.g. `a <=> (b;c) | (d;e)` instead of `a <=> b;c | d;e` and `a <=> true | (d;e)` instead of `a <=> (d;e)`.

In simpagation rules, '`\`' separates the heads of the rule into two parts.

Individual head constraints may be tagged with variables via '`#`', which may be used as identifiers in pragma declarations, for example. Constraint identifiers must be distinct variables, not occurring elsewhere in the heads.

Guards test the applicability of a rule. Guards come in two parts, tell and ask, separated by '`&`'. If the '`&`' operator is not present, the whole guard is assumed to be of the ask type.

Declaratively, a rule relates heads and body *provided the guard is true*. A simplification rule means that the heads are true if and only if the body is true. A propagation rule means that the body is true if the heads are true. A simpagation rule combines a simplification and a propagation rule. The rule `Heads1 \ Heads2 <=> Body` is equivalent to the simplification rule `Heads1, Heads2 <=> Heads1, Body`. However, the simpagation rule is more compact to write, more efficient to execute and has better termination behavior than the corresponding simplification rule, since the constraints comprising `Heads1` will not be removed and inserted again.

31.3.4 How CHR work

Each CHR constraint is associated with all rules in whose heads it occurs by the CHR compiler. Every time a CHR constraint is executed (called) or woken and reconsidered, it checks itself the applicability of its associated CHR by *trying* each CHR. By default, the rules are tried in textual order, i.e. in the order they occur in the defining file. To try a CHR, one of its heads is matched against the constraint. Matching succeeds if the constraint is an instance of the head. If a CHR has more than one head, the constraint store is searched for *partner* constraints that match the other heads. Heads are tried from left to right, except that in simpagation rules, the heads to be removed are tried before the head constraints to be kept (this is done for efficiency reasons). If the matching succeeds, the guard is executed. Otherwise the next rule is tried.

The guard either succeeds or fails. A guard succeeds if the execution of its Ask and Tell parts succeeds and in the ask part no variable that occurs also in the heads was touched or the cause of an instantiation error. The ask guard will fail otherwise. A variable is *touched* if it is unified with a term (including other variables from other constraints) different from itself. Tell guards, on the contrary, are trusted and not checked for that property. If the guard succeeds, the rule applies. Otherwise the next rule is tried.

If the firing CHR is a simplification rule, the matched constraints are removed from the store and the body of the CHR is executed. Similarly for a firing simpagation rule, except that the constraints that matched the heads preceding '\' are kept. If the firing CHR is a propagation rule the body of the CHR is executed without removing any constraints. It is remembered that the propagation rule fired, so it will not fire again with the same constraints if the constraint is woken and reconsidered. If the currently active constraint has not been removed, the next rule is tried.

If the current constraint has not been removed and all rules have been tried, it delays until a variable occurring in the constraint is touched. Delaying means that the constraint is inserted into the constraint store. When a constraint is woken, all its rules are tried again. (This process can be watched and inspected with the CHR debugger, see below.)

31.3.5 Pragmas

Pragmas are annotations to rules and constraints that enable the compiler to generate more specific, more optimized code. A pragma can be a conjunction of the following terms:

`already_in_heads`
: The intention of simplification and simpagation rules is often to combine the heads into a stronger version of one of them. Depending on the strength of the guard, the new constraint may be identical to one of the heads to removed by the rule. This removal followed by addition is inefficient and may even cause termination problems. If the pragma is used, this situation is detected and the corresponding problems are avoided. The pragma applies to all constraints removed by the rule.

`already_in_head(Id)`
: Shares the intention of the previous pragma, but affects only the constraint indicated via *Id*. Note that one can use more than one pragma per rule.

`passive(Id)`
: No code will be generated for the specified constraint in the particular head position. This means that the constraint will not see the rule, it is passive in that rule. This changes the behavior of the CHR system, because normally, a rule can be entered starting from each head constraint. Usually this pragma will improve the efficiency of the constraint handler, but care has to be taken in order not to lose completeness.

  For example, in the handler `leq`, any pair of constraints, say `A leq B, B leq A`, that matches the head `X leq Y , Y leq X` of the `antisymmetry` rule, will also match it when the constraints are exchanged, `B leq A, A leq B`. Therefore it is enough if a currently active constraint enters this rule in the first head only, the second head can be declared to be passive. Similarly for the `idempotence` rule. For this rule, it is more efficient to declare the first head passive, so that the currently active constraint will be removed when the rule fires (instead of removing the older constraint and redoing all the propagation with the currently active constraint). Note that the compiler itself detects the symmetry of the two head constraints in the simplification rule `antisymmetry`, thus it is automatically declared passive and the compiler outputs `CHR eliminated code for head 2 in antisymmetry`.

```
antisymmetry  X leq Y , Y leq X # Id <=> X=Y pragma passive(Id).
idempotence   X leq Y # Id \ X leq Y <=> true pragma passive(Id).
transitivity  X leq Y # Id , Y leq Z ==> X leq Z pragma passive(Id).
```

  Declaring the first head of rule `transitivity` passive changes the behavior of the handler. It will propagate less depending on the order in which the constraints arrive:

```
?- X leq Y, Y leq Z.
X leq Y,
Y leq Z,
X leq Z ?

?- Y leq Z, X leq Y.
Y leq Z,
X leq Y ?
```

```
?- Y leq Z, X leq Y, Z leq X.
Y = X,
Z = X ?
```

The last query shows that the handler is still complete in the sense that all circular chains of leq-relations are collapsed into equalities.

31.3.6 Options

Options parametrise the rule compilation process. Thus they should precede the rule definitions. Example:

```
option(check_guard_bindings, off).
```

The format below lists the names of the recognized options together with the acceptable values. The first entry in the lists is the default value.

`option(debug_compile, [off,on]).`
: Instruments the generated code such that the execution of the rules may be traced (see Section 31.4 [CHR Debugging], page 316).

`option(check_guard_bindings, [on,off]).`
: Per default, for guards of type ask the CHR runtime system makes sure that no variables are touched or the cause of an instantiation error. These checks may be turned off with this option, i.e. all guards are treated as if they were of the tell variety. The option was kept for backward compatibility. Tell and ask guards offer better granularity.

`option(already_in_store, [off,on]).`
: If this option is on, the CHR runtime system checks for the presence of an identical constraint upon the insertion into the store. If present, the attempted insertion has no effect. Since checking for duplicates for all constraints costs, duplicate removal specific to individual constraints, using a few simpagation rules of the following form instead, may be a better solution.

```
Constraint \ Constraint <=> true.
```

`option(already_in_heads, [off,on]).`
: The intention of simplification and simpagation rules is often to combine the heads into a stronger version of one of them. Depending on the strength of the guard, the new constraint may be identical to one of the heads removed by the rule. This removal followed by addition is inefficient and may even cause termination problems. If the option is enabled, this situation is detected and the corresponding problems are avoided. This option applies to all constraints and is provided mainly for backward compatibility. Better grained control can be achieved with corresponding pragmas. (see Section 31.3.5 [CHR Pragmas], page 311).

The remaining options are meant for CHR implementors only:

`option(flatten, [on,off]).`

`option(rule_ordering, [canonical,heuristic]).`

```
option(simpagation_scheme, [single,multi]).
option(revive_scheme, [new,old]).
option(dead_code_elimination, [on,off]).
```

31.3.7 Built-In Predicates

This table lists the predicates made available by the CHR library. They are meant for advanced users, who want to tailor the CHR system towards their specific needs.

`current_handler(?Handler, ?Module)`
: Nondeterministically enumerates the defined handlers with the module they are defined in.

`current_constraint(?Handler, ?Constraint)`
: Nondeterministically enumerates the defined constraints in the form *Functor/Arity* and the handlers they are defined in.

`insert_constraint(+Constraint, -Id)`
: Inserts *Constraint* into the constraint store without executing any rules. The constraint will be woken and reconsidered when one of the variables in *Constraint* is touched. *Id* is unified with an internal object representing the constraint. This predicate only gets defined when a handler and constraints are declared (see Section 31.3.2 [CHR Declarations], page 308).

`insert_constraint(+Constraint, -Id, ?Term)`
: Inserts *Constraint* into the constraint store without executing any rules. The constraint will be woken and reconsidered when one of the variables in *Term* is touched. *Id* is unified with an internal object representing the constraint. This predicate only gets defined when a handler and constraints are declared (see Section 31.3.2 [CHR Declarations], page 308).

`find_constraint(?Pattern, -Id)`
: Nondeterministically enumerates constraints from the constraint store that match *Pattern*, i.e. which are instances of *Pattern*. *Id* is unified with an internal object representing the constraint.

`find_constraint(-Var, ?Pattern, -Id)`
: Nondeterministically enumerates constraints from the constraint store that delay on *Var* and match *Pattern*, i.e. which are instances of *Pattern*. The identifier *Id* can be used to refer to the constraint later, e.g. for removal.

`findall_constraints(?Pattern, ?List)`
: Unifies *List* with a list of `Constraint # Id` pairs from the constraint store that match *Pattern*.

`findall_constraints(-Var, ?Pattern, ?List)`
: Unifies *List* with a list of `Constraint # Id` pairs from the constraint store that delay on *Var* and match *Pattern*.

`remove_constraint(+Id)`
: Removes the constraint *Id*, obtained with one of the previous predicates, from the constraint store.

`unconstrained(?Var)`
: Succeeds if no CHR constraint delays on *Var*. Defined as:

```
unconstrained(X) :-
    find_constraint(X, _, _), !, fail.
unconstrained(_).
```

`notify_constrained(?Var)`
: Leads to the reconsideration of the constraints associated with *Var*. This mechanism allows solvers to communicate reductions on the set of possible values of variables prior to making bindings.

31.3.8 Consulting and Compiling Constraint Handlers

The CHR compilation process has been made as transparent as possible. The user deals with files containing CHR just as with files containing ordinary Prolog predicates. Thus CHR may be consulted, compiled with various compilation modes, and compiled to file (see Chapter 5 [Load Intro], page 51).

31.3.9 Compiler-generated Predicates

Besides predicates for the defined constraints, the CHR compiler generates some support predicates in the module containing the handler. To avoid naming conflicts, the following predicates must not be defined or referred to by user code in the same module:

`verify_attributes/3`

`attribute_goal/2`

`attach_increment/2`

`'attach_F/A'/2`
: for every defined constraint F/A.

`'F/A_N_M_...'/Arity`
: for every defined constraint F/A. N,M is are integers, Arity > A.

For the prime number example that is:

```
attach_increment/2
attach_prime/1/2
attach_primes/1/2
attribute_goal/2
goal_expansion/3
prime/1
prime/1_1/2
```

```
prime/1_1_0/3
prime/1_2/2
primes/1
primes/1_1/2
verify_attributes/3
```

If an author of a handler wants to avoid naming conflicts with the code that uses the handler, it is easy to encapsulate the handler. The module declaration below puts the handler into module `primes`, which exports only selected predicates - the constraints in our example.

```
:- module(primes, [primes/1,prime/1]).

:- use_module(library(chr)).

handler eratosthenes.
constraints primes/1,prime/1.
...
```

31.3.10 Operator Declarations

This table lists the operators as used by the CHR library:

```
:- op(1200, xfx, @).
:- op(1190, xfx, pragma).
:- op(1180, xfx, [==>,<=>]).
:- op(1180, fy,  chr_spy).
:- op(1180, fy,  chr_nospy).
:- op(1150, fx,  handler).
:- op(1150, fx,  constraints).
:- op(1150, fx,  rules).
:- op(1100, xfx, '|').
:- op(1100, xfx, \ ).
:- op(1050, xfx, &).
:- op( 500, yfx, #).
```

31.3.11 Exceptions

The CHR runtime system reports instantiation and type errors for the predicates:

`find_constraint/2`

`findall_constraints/3`

`insert_constraint/2`

`remove_constraint/1`

`notify_constrained/1`

The only other CHR specific runtime error is:

`{CHR ERROR: registering <New>, module <Module> already hosts <Old>}`
: An attempt to load a second handler New into module <Module> already hosting handler <Old> was made.

The following exceptional conditions are detected by the CHR compiler:

`{CHR Compiler ERROR: syntax rule <N>: <Term>}`
: If the N-th <Term> in the file being loaded violates the CHR syntax (see Section 31.3.3 [CHR Syntax], page 309).

`{CHR Compiler ERROR: too many general heads in <Name>}`
: Unspecific heads in definitions like `C \ C <=> true` must not be combined with other heads in rule <Name>.

`{CHR Compiler ERROR: bad pragma <Pragma> in <Name>}`
: The pragma <Pragma> used in rule <Name> does not qualify. Currently this only happens if <Pragma> is unbound.

`{CHR Compiler ERROR: found head <F/A> in <Name>, expected one of: <F/A list>}`
: Rule <Name> has a head of given F/A which is not among the defined constraints.

`{CHR Compiler ERROR: head identifiers in <Name> are not unique variables}`
: The identifiers to refer to individual constraints (heads) via '#' in rule <Name> do not meet the indicated requirements.

`{CHR Compiler ERROR: no handler defined}`
: CHR specific language elements, declarations or rules for example, are used before a handler was defined. This error is usually reported a couple of times, i.e. as often as there are CHR forms in the file expecting the missing definition.

`{CHR Compiler ERROR: compilation failed}`
: Not your fault. Send us a bug report.

31.4 Debugging CHR Programs

Use `option(debug_compile,on)` preceding any rules in the file containing the handler to enable CHR debugging. The CHR debugging mechanism works by instrumenting the code generated by the CHR compiler. Basically, the CHR debugger works like the Prolog debugger. The main differences are: there are extra ports specific to CHR, and the CHR debugger provides no means for the user to change the flow of control, i.e. there are currently no *retry* and *fail* options available.

31.4.1 Control Flow Model

The entities reflected by the CHR debugger are constraints and rules. Constraints are treated like ordinary Prolog goals with the usual ports: `[call,exit,redo,fail]`. In addition, constraints may get inserted into or removed from the constraint store (ports: `insert,remove`), and stored constraints containing variables will be woken and reconsidered (port: `wake`) when variables are touched.

The execution of a constraint consists of trying to apply the rules mentioning the constraint in their heads. Two ports for rules reflect this process: At a `try` port the active constraint matches one of the heads of the rule, and matching constraints for the remaining heads of the rule, if any, have been found as well. The transition from a `try` port to an `apply` port takes place when the guard has been successfully evaluated, i.e. when the rule commits. At the `apply` port, the body of the rule is just about to be executed. The body is a Prolog goal transparent to the CHR debugger. If the rule body contains CHR constraints, the CHR debugger will track them again. If the rules were consulted, the Prolog debugger can be used to study the evaluations of the other predicates in the body.

31.4.2 CHR Debugging Predicates

The following predicates control the operation of the CHR debugger:

`chr_trace`
: Switches the CHR debugger on and ensures that the next time control enters a CHR port, a message will be produced and you will be asked to interact.

  At this point you have a number of options. See Section 31.4.5 [CHR Debugging Options], page 320. In particular, you can just type `CR` (Return) to *creep* (or single-step) into your program. You will notice that the CHR debugger stops at many ports. If this is not what you want, the predicate `chr_leash` gives full control over the ports at which you are prompted.

`chr_debug`
: Switches the CHR debugger on and ensures that the next time control enters a CHR port with a spy-point set, a message will be produced and you will be asked to interact.

`chr_nodebug`
: Switches the CHR debugger off. If there are any spy-points set then they will be kept.

`chr_notrace`
: Equivalent to `chr_nodebug`.

`chr_debugging`
: Prints onto the standard error stream information about the current CHR debugging state. This will show:

  1. Whether the CHR debugger is switched on.
  2. What spy-points have been set (see below).
  3. What mode of leashing is in force (see below).

`chr_leash(+`*Mode*`)`

The leashing mode is set to *Mode*. It determines the CHR ports at which you are to be prompted when you *creep* through your program. At unleashed ports a tracing message is still output, but program execution does not stop to allow user interaction. Note that the ports of spy-points are always leashed (and cannot be unleashed). *Mode* is a list containing none, one or more of the following port names:

`call` Prompt when a constraint is executed for the first time.

`exit` Prompt when the constraint is successfully processed, i.e. the applicable rules have applied.

`redo` Prompt at subsequent exits generated by non-deterministic rule bodies.

`fail` Prompt when a constraint fails.

`wake` Prompt when a constraint from the constraint store is woken and reconsidered because one of its variables has been touched.

`try` Prompt just before the guard evaluation of a rule, after constraints matching the heads have been found.

`apply` Prompt upon the application of a rule, after the successful guard evaluation, when the rule commits and fires, just before evaluating the body.

`insert` Prompt when a constraint gets inserted into the constraint store, i.e. after all rules have been tried.

`remove` Prompt when a constraint gets removed from the constraint store, e.g. when a simplification rule applies.

The initial value of the CHR leashing mode is `[call,exit,fail,wake,apply]`. Predefined shortcuts are:

`chr_leash(none)`, `chr_leash(off)`
To turn leashing off.

`chr_leash(all)`
To prompt at every port.

`chr_leash(default)`
Same as `chr_leash([call,exit,fail,wake,apply])`.

`chr_leash(call)`
No need to use a list if only a singular port is to be leashed.

31.4.3 CHR Spy-points

For CHR programs of any size, it is clearly impractical to creep through the entire program. *Spy-points* make it possible to stop the program upon an event of interest. Once there, one can set further spy-points in order to catch the control flow a bit further on, or one can start creeping.

Setting a spy-point on a constraint or a rule indicates that you wish to see all control flow through the various ports involved, except during skips. When control passes through any port with a spy-point set on it, a message is output and the user is asked to interact. Note that the current mode of leashing does not affect spy-points: user interaction is requested on *every* port.

Spy-points are set and removed by the following predicates, which are declared as prefix operators:

`chr_spy` *Spec*
: Sets spy-points on constraints and rules given by *Spec*, which is is of the form:

  _ *(variable)*
  : denoting all constraints and rules, or:

  constraints Cs
  : where *Cs* is one of

    _ *(variable)*
    : denoting all constraints

    C,...,C denoting a list of constraints *C*

    Name denoting all constraints with this functor, regardless of arity

    Name/Arity
    : denoting the constraint of that name and arity

  rules Rs where *Rs* is one of:

    _ *(variable)*
    : denoting all rules

    R,...,R denoting a list of rules *R*

    Name where *Name* is the name of a rule in any handler.

    already_in_store
    : The name of a rule implicitly defined by the system when the option `already_in_store` is in effect.

    already_in_heads
    : The name of a rule implicitly defined by the system when the option `already_in_heads` or the corresponding pragmas are in effect.

    Handler:Name
    : where *Handler* is the name of a constraint handler and *Name* is the name of a rule in that handler

  Examples:

```
| ?- chr_spy rules rule(3), transitivity, already_in_store.
| ?- chr_spy constraints prime/1.
```

If you set spy-points, the CHR debugger will be switched on.

`chr_nospy` *Spec*
: Removes spy-points on constraints and rules given by *Spec*, where *Spec* is of the form as described for `chr_spy` *Spec*. There is no `chr_nospyall/0`. To remove all CHR spy-points use `chr_nospy _`.

The options available when you arrive at a spy-point are described later. See Section 31.4.5 [CHR Debugging Options], page 320.

31.4.4 CHR Debugging Messages

All trace messages are output to the standard error stream. This allows you to trace programs while they are performing file I/O. The basic format is as follows:

```
S 3   1 try     eratosthenes:absorb(10) @ prime(9)#<c4>, prime(10)#<c2> ?
```

S is a spy-point indicator. It is printed as ' ' if there is no spy-point, as '`r`', indicating that there is a spy-point on this rule, or as '`c`' if one of the involved constraints has a spy-point.

The first number indicates the current depth of the execution; i.e. the number of direct *ancestors* the currently active constraint has.

The second number indicates the head position of the currently active constraint at rule ports.

The next item tells you which port is currently traced.

A constraint or a matching rule are printed next. Constraints print as `Term#Id`, where *Id* is a unique identifier pointing into the constraint store. Rules are printed as `Handler:Name @`, followed by the constraints matching the heads.

The final '`?`' is the prompt indicating that you should type in one of the debug options (see Section 31.4.5 [CHR Debugging Options], page 320).

31.4.5 CHR Debugging Options

This section describes the options available when the system prompts you after printing out a debugging message. Most of them you know from the standard Prolog debugger. All the options are one letter mnemonics, some of which can be optionally followed by a decimal integer. They are read from the standard input stream up to the end of the line (Return, `<CR>`). Blanks will be ignored.

The only option which you really have to remember is '`h`'. This provides help in the form of the following list of available options.

```
CHR debugging options:
   <cr>   creep            c      creep
    l     leap
    s     skip             s <i>  skip (ancestor i)
    g     ancestors
    &     constraints      & <i>  constraints (details)
    n     nodebug          =      debugging
    +     spy this
    -     nospy this       .      show rule
    <     reset printdepth < <n>  set printdepth
    a     abort            b      break
    ?     help             h      help
```

`c`
`<CR>` *creep* causes the debugger to single-step to the very next port and print a message. Then if the port is leashed, the user is prompted for further interaction. Otherwise, it continues creeping. If leashing is off, creep is the same as *leap* (see below) except that a complete trace is printed on the standard error stream.

`l` *leap* causes the debugger to resume running your program, only stopping when a spy-point is reached (or when the program terminates). Leaping can thus be used to follow the execution at a higher level than exhaustive tracing.

`s`
`s i` *skip* over the entire execution of the constraint. That is, you will not see anything until control comes back to this constraint (at either the `exit` port or the `fail` port). This includes ports with spy-points set; they will be masked out during the skip. The command can be used with a numeric argument to skip the execution up to and including the ancestor indicated by the argument. Example:

```
            ...
            4   - exit    prime(8)#<c6> ? g
      Ancestors:
            1   1 apply   eratosthenes:rule(2) @ primes(10)#<c1>
            2   1 apply   eratosthenes:rule(2) @ primes(9)#<c3>
            3   1 apply   eratosthenes:rule(2) @ primes(8)#<c5>
            4   - call    prime(8)#<c6>

            4   - exit    prime(8)#<c6> ? s 2
            2   - exit    primes(9)#<c3> ?
```

`g` *print ancestors* provides you with a list of ancestors to the currently active constraint, i.e. all constraints not yet exited that led to the current constraint in the derivation sequence. The format is the same as with trace messages. Constraints start with `call` entries in the stack. The subsequent application of a rule replaces the call entry in the stack with an `apply` entry. Later the constraint shows again as `redo` or `fail` entry. Example:

```
      0   - call    primes(10)#<c1> ?
      1   1 try     eratosthenes:rule(2) @ primes(10)#<c1> ? g

Ancestors:
      1   - call    primes(10)#<c1>

      1   1 try     eratosthenes:rule(2) @ primes(10)#<c1> ?
      1   1 apply   eratosthenes:rule(2) @ primes(10)#<c1> ?
      1   - call    prime(10)#<c2> ?
      2   - insert  prime(10)#<c2>
      2   - exit    prime(10)#<c2> ? g

Ancestors:
      1   1 apply   eratosthenes:rule(2) @ primes(10)#<c1>
      2   - call    prime(10)#<c2>
```

`&` *print constraints* prints a list of the constraints in the constraint store. With a numeric argument, details relevant primarily to CHR implementors are shown.

`n` *nodebug* switches the CHR debugger off.

`=` *debugging* outputs information concerning the status of the CHR debugger as via `chr_debugging/0`

`+` *spy this* sets a spy-point on the current constraint or rule.

`-` *nospy this* removes the spy-point from the current constraint or rule, if it exists.

`.` *show rule* prints the current rule instantiated by the matched constraints. Example:

```
8   1 apply   era:absorb(8) @ prime(4)#<c14> \ prime(8)#<c6> ? .

absorb(8) @
 prime(4)#<c14> \
   prime(8)#<c6> <=>

   8 mod 4=:=0
   |
   true.
```

`<`

`< n` While in the debugger, a *printdepth* is in effect for limiting the subterm nesting level when printing rules and constraints. The limit is initially 10. This command, without arguments, resets the limit to 10. With an argument of *n*, the limit is set to *n*, treating 0 as infinity.

`a` *abort* calls the built-in predicate `abort/0`.

`b` *break* calls the built-in predicate `break/0`, thus putting you at a recursive top-level. When you end the break (entering `^D`) you will be re-prompted at the port at which you broke. The CHR debugger is temporarily switched off as you call the break and will be switched on again when you finish the break and go back to the old execution. Any changes to the CHR leashing or to spy-points during the break will remain in effect.

`?`
`h` *help* displays the table of options given above.

31.5 Programming Hints

This section gives you some programming hints for CHR. For maximum efficiency of your constraint handler, see also the previous subsections on declarations and options.

Constraint handling rules for a given constraint system can often be derived from its definition in formalisms such as inference rules, rewrite rules, sequents, formulas expressing axioms and theorems. CHR can also be found by first considering special cases of each constraint and then looking at interactions of pairs of constraints sharing a variable. Cases that do not occur in the application can be ignored.

It is important to find the right *granularity* of the constraints. Assume one wants to express that *n* variables are different from each other. It is more efficient to have a single constraint `all_different(List_of_n_Vars)` than *n*n* inequality constraints between each pair of different variables. However, the extreme case of having a single constraint modeling the whole constraint store will usually be inefficient.

Starting from an executable specification, the rules can then be refined and adapted to the specifics of the application. Efficiency can be improved by weakening the guards to perform simplification as early as needed and by strengthening the guards to do the *just right* amount of propagation. Propagation rules can be expensive, because no constraints are removed.

The more heads a rule has, the more expensive it is. *Rules with several heads* are more efficient, if the heads of the rule share a variable (which is usually the case). Then the search for a partner constraint has to consider less candidates. In the current implementation, constraints are indexed by their functors, so that the search is only performed among the constraints containing the shared variable. Moreover, two rules with identical (or sufficiently similar) heads can be merged into one rule so that the search for a partner constraint is only performed once instead of twice.

As *guards* are tried frequently, they should be simple *tests* not involving side-effects. Head matching is more efficient than explicitly checking equalities in the ask-part of the guard. In the tell part of a guard, it should be made sure that variables from the head are never touched (e.g. by using `nonvar` or `ground` if necessary). For efficiency and clarity reasons, one should also avoid using constraints in guards. Besides conjunctions, disjunctions are allowed in the guard, but they should be used with care. The use of other control built-in predicates in the guard is discouraged. Negation and if-then-else in the ask part of a guard can give wrong results, since e.g. failure of the negated goal may be due to touching its variables.

Several handlers can be used simultaneously if they do not share constraints with the same name. The implementation will not work correctly if the same constraint is defined in rules of different handlers that have been compiled separately. In such a case, the handlers must be merged *by hand*. This means that the source code has to be edited so that the rules for the shared constraint are together (in one module). Changes may be necessary (like strengthening guards) to avoid divergence or loops in the computation.

31.6 Constraint Handlers

The CHR library comes with plenty of constraint handlers written in CHR:

'`arc.pl`' classical arc-consistency over finite domains

'`bool.pl`' simple Boolean constraints

'`cft.pl`' feature term constraints according to the CFT theory

'`domain.pl`'
finite domains over arbitrary ground terms and interval domains over integers and reals, but without arithmetic functions

'`gcd.pl`' elegant two-liner for the greatest common divisor

'`interval.pl`'
straightforward interval domains over integers and reals, with arithmetic functions

'`kl-one.pl`'
terminological reasoning similar to KL-ONE or feature trees

'`leq.pl`' standard introductory CHR example handler for less-than-or-equal

'`list.pl`' equality constraints over concatenations of lists (or strings)

'`math-elim.pl`'
solves linear polynomial equations and inequations using variable elimination, several variations possible

'`math-fougau.pl`'
solves linear polynomial equations and inequations by combining variable elimination for equations with Fourier's algorithm for inequations, several variations possible

'`math-gauss.pl`'
a straightforward, elegant implementation of variable elimination for equations in one rule

'`minmax.pl`'
simple less-than and less-than-or-equal ordering constraints together with minimum and maximum constraints

'`modelgenerator.pl`'
example of how to use CHR for model generation in theorem proving

'`monkey.pl`'
classical monkey and banana problem, illustrates how CHR can be used as a fairly efficient production rule system

'`osf.pl`' constraints over order sorted feature terms according to the OSF theory

'`pathc.pl`'
the most simple example of a handler for path consistency - two rules

'`primes.pl`'
: elegant implementations of the sieve of Eratosthenes reminiscent of the chemical abstract machine model, also illustrates use of CHR as a general purpose concurrent constraint language

'`scheduling.pl`'
: simple classical constraint logic programming scheduling example on building a house

'`tarski.pl`'
: most of Tarski's axiomatization of geometry as constraint system

'`term.pl`' Prolog term manipulation built-in predicates `functor/3, arg/3, =../2` as constraints

'`time-pc.pl`'
: grand generic handler for path-consistency over arbitrary constraints, load via '`time.pl`' to get a powerful solver for temporal constraints based on Meiri's unifying framework. '`time-rnd.pl`' contains a generator for random test problems.

'`time-point.pl`'
: quantitative temporal constraints over time points using path-consistency

'`tree.pl`' equality and disequality over finite and infinite trees (terms)

'`type.pl`' equalities and type constraints over finite and infinite trees (terms)

You can consult or compile a constraint handler from the CHR library using e.g.:

```
?- [library('chr/examples/gcd')].
?- compile(library('chr/examples/gcd')).
```

If you want to learn more about the handlers, look at their documented source code.

In addition, there are files with example queries for some handlers, their file name starts with '`examples-`' and the file extension indicates the handler, e.g. '`.bool`':

```
examples-adder.bool
examples-benchmark.math
examples-deussen.bool
examples-diaz.bool
examples-fourier.math
examples-holzbaur.math
examples-lim1.math
examples-lim2.math
examples-lim3.math
examples-puzzle.bool
examples-queens.bool
examples-queens.domain
examples-stuckey.math
examples-thom.math
```

31.7 Backward Compatibility

In this section, we discuss backward compatibility with the CHR library of Eclipse Prolog.

1. The restriction on at most two heads in a rule has been abandoned. A rule can have as many heads as you like. Note however, that searching for partner constraints can be expensive.
2. By default, rules are compiled in textual order. This gives the programmer more control over the constraint handling process. In the Eclipse library of CHR, the compiler was optimizing the order of rules. Therefore, when porting a handler, rules may have to be reordered. A good heuristic is to prefer simplification to simpagation and propagation and to prefer rules with single heads to rules with several heads. Instead of manually rearranging an old handler one may also use the following combination of options to get the corresponding effect:

   ```
   option(rule_ordering,heuristic).
   option(revive_scheme,old).
   ```
3. For backward compatibility, the `already_in_store`, `already_in_head` and `guard_bindings` options are still around, but there are CHR syntax extensions: Section 31.3.3 [CHR Syntax], page 309 and pragmas Section 31.3.5 [CHR Pragmas], page 311 offering better grained control.
4. The Eclipse library of CHR provided automatic built-in labeling through the `label_with` declaration. Since it was not widely used and can be easily simulated, built-in labeling was dropped. The same effect can be achieved by replacing the declaration `label_with Constraint if Guard` by the simplification rule `chr_labeling, Constraint <=> Guard | Constraint', chr_labeling` and by renaming the head in each clause `Constraint :- Body` into `Constraint' :- Body` where `Constraint'` is a new predicate. Efficiency can be improved by declaring `Constraint` to be passive: `chr_labeling, Constraint#Id <=> Guard | Constraint', chr_labeling pragma passive(Id)`. This translation will not work if `option(already_in_heads,on)`. In that case use e.g. `chr_labeling(_), Constraint <=> Guard | Constraint', chr_labeling(_)` to make the new call to `chr_labeling` differ from the head occurrence.
5. The set of built-in predicates for advanced CHR users is now larger and better designed. Also the debugger has been improved. The Opium debugging environment is not available in SICStus Prolog.

32 Prolog Objects

Prolog Objects is an extension to SICStus Prolog for flexible structuring, sharing and reuse of knowledge in large logic programming applications. Prolog Objects enhances Prolog with an expressive and efficient object-oriented programming component.

Prolog Objects is based on the notion of prototypes. In object-oriented programming a *prototype* is an object that represents a typical behavior of a certain concept. A prototype can be used as is or as a model to construct other objects that share some of the characteristics of the prototypical object. These specialized objects can themselves become prototypes used to construct other objects and so forth. The basic mechanism for sharing is by *inheritance* and *delegation*. Inheritance is known for most readers. By using the delegation mechanism an object can forward a message to another object to invoke a method defined by the recipient but interpreted in the context of the sender.

In Prolog Objects, an *object* is a named collection of predicate definitions. In this sense an object is similar to a Prolog module. The object system can be seen as an extension of SICStus Prolog's module system. In addition an object may have attributes that are modifiable. Predicate definitions belonging to an object are called *methods*. So, an object is conceptually a named collection of methods and attributes. Some of the methods defined for an object need not be stored explicitly within the object, but are rather shared with other objects by the inheritance mechanism.

The Object system allows objects to be defined in a file, or dynamically created during the execution of a program. Objects defined in a file are integrated into SICStus Prolog in a way similar to definite clause grammars. That is to say, objects have a specific syntax as Prolog terms, and can be loaded and expanded into Prolog code. When an object is created, during load-time, or run-time, it inherits the methods and attributes of its prototypical object(s). Objects defined in a file can be either *static* or *dynamic*. Also, methods can be either dynamic or static. these properties are inherited by sub-objects. Objects created during execution are dynamic.

The inheritance mechanism is implemented using the importation mechanism of the module system. The default inheritance is an *inheritance by overriding* mechanism, which means that if a method is defined locally, and the same method is defined in a super-object, then the clauses of the super-method are not part of the definition of the local one. As usual in Prolog, methods can be non-determinately defined, and alternative answers can be retrieved through backtracking. Using the delegation mechanism, other methods for knowledge sharing can be implemented by the user. In Objects, there is an initial prototypical proto-object called `object`, from which other objects may be constructed, directly or indirectly.

32.1 Getting Started

To load the Prolog Objects library, enter the query:

```
| ?- use_module(library(objects)).
```

Prolog Objects defines some new infix and prefix operators, and redefines some of the built-in ones. The following operators become installed:

```
:- op(1200, xfy, [ & ]).
:- op(1198, xfx, [ :- ]).
:- op(1198,  fx, [ :- ]).
:- op(550,  xfx, [ ::, <: ]).
:- op(550,   fx, [ ::, <: ]).
```

32.2 Declared Objects

Declared objects are created when the files defining them are loaded into the system.

32.2.1 Object Declaration

An object *object-identifier* is declared by writing it in the following form:

object-identifier :: `{`
sentence-1 &
sentence-2 &
:
sentence-n
`}`.

where *object-identifier* is a Prolog term that is either an atom or a compound term of the form *functor(V1,...,Vn)*, where *V1,...,Vn* are distinct variables. The object body consists of a number of *sentences*, possibly none, surrounded by braces, where each sentence is either a *method-directive*, to be executed when the object is created, or a *method-clause*. A *method* is a number of method-clauses with the same principal functor. A method-clause has a clausal syntax similar to that of Prolog, but instead of usual predicate calls in the body of a clause there are *method-calls*. Ordinary Prolog goals are also allowed in a prefixed form, using ':' as a prefix. A method-directive is a directive which contains method-calls.

All sentences are subject to term expansion (see Section 7.1.2 [Definite], page 76, built-in `expand_term/2`) before further processing, so in particular definite clause grammar syntax can be used in method-clauses. In addition, before `expand_term/2`, sentences are expanded by the predicate `user:method_expansion/3`.

`method_expansion(+`*Term1*`,+`*ObjectIdentifier*`,?`*Term2*`)`
`user:method_expansion(+`*Term1*`,+`*ObjectIdentifier*`,?`*Term2*`)`
: *A hook predicate*, which defines transformations on methods similarly as `user:term_expansion/(2,4)`. At the end of an object definition, `user:method_expansion/3` is called with `end_of_object`.

32.2.2 Method Declarations

Method-clauses are declared similarly to Prolog clauses. Thus a method-clause can be either a unit-clause or a non-unit-clause. We also allow a default catch-all method-clause as the last clause in an object body. The catch-all clause has as its head a Prolog variable, in order to match messages that are not previously defined or inherited in the object. It can be used to implement alternative inheritance mechanisms.

Goals in the body of a non-unit clause have the normal control structures of Prolog:

:P`,` *:Q*
: Conjunction

:P`;` *:Q*
: Disjunction

`!`
: Cut

`\+` *:P*
: Negation

:P `->` *:Q*
:P `->` *:Q* `;` *:R*
`if(`*:P*`,` *:Q*`,` *:R*`)`
: If-then[-else]

?A `=` *?B*
: Unification

Atomic goals in the body of a method-clause may be one of the following:

:goal
: to call the Prolog predicate *goal* in the source module.

m`:`*goal*
: to call the Prolog predicate *goal* in module *m*.

goal
: to send the message *goal* to the object *Self*.

`::`*goal*
: to send the message *goal* to a method that may be defined locally or inherited by the object.

`<:`*goal*
: to delegate the message *goal* to a method that may be defined locally or inherited by the object.

object`::`*goal*
: to send the message *goal* to object *object*.

object`<:`*goal*
: to delegate the message *goal* to object *object*.

Message sending and delegation will be explained later (see Section 32.3 [Obj Self], page 332).

The following is a definition for the object `list_object`. It is constructed from three methods: `append/3`, `member/2`, and `length/2`. Note that the calls to `append/3` and `length/2` are to the local definition, whereas the `member/2` call is to the predicate imported from the Prolog library module `lists`.

```
list_object :: {
        :- :use_module(library(lists), [append/3,member/2]) &

        append([], L, L) &
        append([X|L1], L2, [X|L3]) :-
                :: append(L1, L2, L3) &

        member(X, L) :-
                :member(X,L) &

        length([], 0) &
        length([_|L], N) :-
                :: length(L, N1),
                :(N is N1+1)
        }.
```

The following object `apt_1` could be part of a larger database about free apartments in a real-estate agency:

```
apt_1 :: {
        super(apartment) &

        street_name('York') &
        street_number(100) &
        wall_color(white) &
        floor_surface(wood)
        }.
```

Another way to define `apt_1` is by using attributes. These can be retrieved and modified efficiently by the methods `get/1` and `set/1` respectively.

```
apt_1 :: {
        super(apartment) &

        attributes([
               street_name('York'),
               street_number(100),
               wall_color(white),
               floor_surface(wood)])
        }.
```

32.2.3 Generic Objects for Easy Reuse

Defining objects for easy reuse is a very important property for reducing the cost of large projects. One important technique is to define prototypes in a parameterized way, so that various instantiations of a prototype correspond to different uses. Parameterized or *generic objects* have been used for this purpose in other object-oriented systems. An object-identifier can be a compound term. The arguments of the term are parameters that are visible in the object-body. Here we show one example. Other examples and techniques that use this facility has been investigated extensively in [McCabe 92].

The following is an object `sort` that sorts lists of different types. `sort` has a parameter that defines the type of the elements of the list. Notice that *Type* is visible to all methods in the body of `sort`, and is used in the method `partition/4`. In the query, we use `sort(rat)` to sort a list of terms denoting rational numbers. We must therefore define a `rat` object and its `<` method also:

```
rat :: {
        (P/Q < R/S) :- :(P*S < Q*R)
        }.

sort(Type) :: {
        :- :use_module(library(lists), [append/3]) &

        qsort([], []) &
        qsort([P|L], S) :-
                partition(L, P, Small, Large),
                qsort(Small, S0),
                qsort(Large, S1),
                :append(S0, [P|S1], S) &

        partition([], _P, [], []) &
        partition([X|L1], P, Small, Large) :-
                (   Type :: (X < P) ->
                    Small = [X|Small1], Large = Large1
                ;   Small = Small1, Large = [X|Large1]
                ),
                partition(L1, P, Small1, Large1)
        }.

| ?- sort(rat) :: qsort([23/3, 34/11, 45/17], L).

L = [45/17,34/11,23/3]
```

Parameterized objects are interesting in their own right in Prolog even if one is not interested in the object-oriented paradigm. They provide global context variables in a Prolog program without having to add such variables as additional context arguments to each clause that potentially uses the context.

32.3 Self, Message Sending, and Message Delegation

In Prolog Objects, each method is executed in the context of an object. This object may not be the static object where the method is declared. The current contextual object is used to determine dynamically which attributes are accessed, and which methods are called. This leads to a mechanism known as dynamic binding. This object can be retrieved using the universal method `self(`*S*`)`, where *S* will be bound to the current contextual object.

When a message is sent to an object, the corresponding method will be executed in the context of the target object. A message delegated to an object will invoke a method that is executed in the context of the message-delegation operation.

object `::` *message*
`::` *message* Message sending. Sends *message* to *object*, setting *Self* of the recipient to the recipient, i.e. *object*. If *object* is omitted, the recipient is the object in which the goal textually appears.

object `<:` *message*
`<:` *message*
Message delegation. Sends *message* to *object*, setting *Self* of the recipient to *Self* of the sender. If *object* is omitted, the recipient is the object in which the goal textually appears. *Delegation preserves Self*.

The following objects `physical_object`, `a`, and `b` are written using the default notations for sending and delegation, hiding the contextual variable *Self*:

```
physical_object :: {
        volume(50) &
        density(100) &
        weight(X) :-
                volume(V),
                density(D),
                :(X is V*D)
        }.

a :: {
        volume(5) &
        density(10) &

        Method :-
                physical_object <: Method
        }.

b :: {
        volume(5) &
        density(10) &

        Method :-
                physical_object :: Method
```

```
        }.
```

Notice that the difference between the objects `a` and `b` is that `a` *delegates* any message except `volume(_)` and `density(_)` to `physical_object` while `b` *sends* the message to `physical_object`. We may now ask

```
| ?- a :: weight(X), b :: weight(Y).

X = 50
Y = 5000
```

To get hold of the current contextual object, the universal method `self(`S`)` is provided. Another way to send a message to *Self* is to use the constant `self`. So the following two alternative definition of `physical_object` are equivalent to the previous one:

```
physical_object :: {
        volume(50) &
        density(100) &
        weight(X) :-
                self(S),
                S::volume(V),
                S::density(D),
                :(X is V*D)
        }.

physical_object :: {
        volume(50) &
        density(100) &
        weight(X) :-
                self::volume(V),
                self::density(D),
                :(X is V*D)
        }.
```

32.4 Object Hierarchies, Inheritance, and Modules

The Prolog Objects system implements a default inheritance mechanism. By declaring within an object which objects are super-objects, the hierarchy of objects are maintained. The system also maintains for each object its immediate sub-objects (i.e. immediate children). Each object may also call Prolog predicates. At the top of the hierarchy, the proto-object `object` provides various services for other objects. If `object` is not used at the top of the hierarchy many services will not be available for other objects (check what methods are available in `object` by sending the message `method/1` to `object`).

32.4.1 Inheritance

Immediate super-objects are declared by defining the method `super/2` within the object. (Any definition `super(`*Super*`)` is transformed to `super(`*Super*`,[])`). The objects declared by `super/2` are the immediate objects from which a method is inherited if not defined within the object. This implies that the inheritance mechanism is an overriding one. One could possibly have a union inheritance, whereby all clauses defining a method are collected from the super hierarchy and executed in a Prolog fashion. This can easily be programmed in Prolog Objects, using delegation to super objects.

The following example shows some objects used for animal classification.

```
animal :: {}.

bird :: {
        super(animal) &
        skin(feather) &
        habitat(tree) &
        motions(fly)
        }.

penguin :: {
        super(bird) &
        habitat(land) &
        motions(walk) &
        motions(swim) &
        size(medium)
        }.

| ?- penguin :: motions(M).
M = walk ;
M = swim ;
no

| ?- penguin :: skin(S).
S = feather ;
no
```

The following is an example of multiple inheritance: an object `john` is both a sportsman and a professor:

```
john :: {
        super(sportsman) &
        super(professor) &
        :
        }.
```

Inheritance will give priority to the super-objects by the order defined in the `super/2` method. Therefore in the above example John's characteristics of being a sportsman will dominate those

of being professor. Other kinds of hierarchy traversal can be programmed explicitly using the delegation mechanism.

32.4.2 Differential Inheritance

It is possible to be selective about what is inherited by using the method `super/2`. Its first argument is the super object, and its second is a list of the methods that will not be inherited from the super object.

32.4.3 Use of Modules

In Prolog Objects, the visible predicates of the source module (context) for the object definition may be called in the body of a method. (The `:` prefix is used to distinguish such calls from method calls.) Any (`:` prefixed) directives occurring among the method-clauses are also executed in the same source module. For example, to import into the source module and call the public predicates of a module, the built-in predicate `use_module/2` and its variants may be used:

```
some_object :: {
        :- :use_module(library(lists), [append/3]) &
        double_list(X, XX) :- :append(X,X,XX)
        }.
```

32.4.4 Super and Sub

Two methods provided by the initial object `object` are `super/1` and `sub/1`. (Note that any definition of `super/1`, except the one in `object`, is transformed to `super/2`). `super/1` if sent to an object will return the immediate parents of the object. `sub/1` will return the immediate children of the object if any. It is important to note that this service is provided only for objects that have `object` as their initial ancestor.

```
| ?- john :: super(S), S :: sub(john).
S = sportsman ;
S = professor ;
no
```

The `sub/1` property allows programs to traverse object hierarchies from a root object `object` down to the leaves.

32.4.5 The Keyword Super

To be able to send or delegate messages to the super-objects in a convenient way while following the inheritance protocol, the keyword `super` is provided. The calls:

```
super :: method, or
super <: method
```

mean: send or delegate (respectively) *method* to the super-objects according to the inheritance protocol. A simple example illustrates this concept: assume that `john` in the above example has three id-cards, one stored in his sportsman prototype identifying the club he is member of, one stored in his professor prototype identifying the university he works in, and finally one stored locally identifying his social-security number. Given the following methods in the object `john`:

```
m1(X) :-
        super <: id_card(X) &
m2(X) :-
        super(S), S <: id_card(X) &
```

one may ask the following:

```
| ?- john :: m1(X).
     % will follow the default inheritance and returns:
X = johns_club ;

| ?- john :: m2(X).
     % will backtrack through the possible supers returning:
X = johns_club ;
X = johns_university ;
```

32.4.6 Semantic Links to Other Objects

Some object-oriented languages have syntactic constructs for redirecting the inheritance chain for certain methods to completely other objects which are not defined in the object's inheritance hierarchy. This is not needed in Prolog Objects due to delegation. Assume that the method `m/`*n* is linked to object `some_object`, we just add a method for this:

```
m(X1, ..., Xn) :- some_object <: m(X1, ..., Xn) &
```

32.4.7 Dynamically Declared Objects

When an object is declared and compiled into Prolog Objects, its methods cannot be changed during execution. Such an object is said to be *static*. To be able to update any method in an object, the object has to be declared *dynamic*. There is one exception, the inheritance hierarchy declared by `super/(1-2)` cannot be changed. By including the fact `dynamic` as part of the object body, the object becomes dynamic:

```
dynamic_object :: {
        dynamic &
        :
        }.
```

32.4.8 Dynamic Methods

To be able to change a method with functor F and arity N in a static object, the method has to be declared dynamic by storing the following fact in the object:

```
some_object :: {
        dynamic F/N &
        :
        }.
```

Each book in a library can be represented as an object, in which the name of the book is stored, the authors, and a borrowing history indicating when a book is borrowed and when it is returned. A history item may have the form `history_item(`*Person*`,`*Status*`,`*Date*`)` where *Status* is either `borrowed` or `returned`, and *Date* has the form YY-MM-DD, for YY year, MM month, DD day.

A typical book `book_12` could have the following status. Note that `history_item/3` is dynamic:

```
book_12 :: {
        super(book) &

        title('The Art of Prolog') &
        authors(['Leon Sterling', 'Ehud Shapiro']) &

        dynamic history_item/3 &

        history_item('Dan Sahlin', returned, 92-01-10) &
        history_item('Dan Sahlin', borrowed, 91-06-10) &
        :
        }.
```

Dynamic methods that are stored in an object can be updated, as in usual Prolog programs, by sending `assert` and `retract` messages directly to the object.

For example, to borrow a book the following method could be defined in the object `book`. We assume that the top most `history_item` fact is the latest transaction, and there is an object `date` from which we can get the current date.

```
borrow(Person) :-
        history_item(_Person0, Status, _Date0), !,
        (   Status = returned ->
            date::current(Date),
            asserta(history_item(Person, borrowed, Date))
        ;   :display('book not available'), :ttynl
        ) &
```

32.4.9 Inheritance of Dynamic Behavior

When an object is created, it will inherit from its parents their dynamic behavior. Methods that are declared dynamic in a parent, will be copied into the object, and its dynamic behavior preserved.

```
a:: {
        super(object) &
        dynamic p/1 &
        p(1) &
        p(2)
    }

b :: {
        super(a)
    }

| ?- b::p(X).
X = 1 ? ;
X = 2 ? ;
no
| ?- b::asserta(p(3)).
yes
| ?- b::p(X).
X = 3 ? ;
X = 1 ? ;
X = 2 ? ;
no
```

Notice that by redeclaring a method to be dynamic in a sub-object, amounts to redefining the method, and overriding of the parent definition will take effect.

```
c :: {
        super(a) &
        dynamic p/1
    }

| ?- c::p(X).
no
```

32.5 Creating Objects Dynamically

As with dynamically declared objects, the full flexibility of Prolog Objects is achieved when objects are created at runtime. Anything, except the inheritance hierarchy, can be changed: methods can be added or deleted. The services for object creation, destruction, and method modification are defined in the proto-object `object`.

32.5.1 Object Creation

+*SomeObject* :: `new(`*?NewObject*`)`
: *NewObject* is created with *SomeObject* as super. *NewObject* could be an atom, variable, or compound term whose arguments are distinct variables.

+*SomeObject* :: `new(`*?NewObject*`,+`*Supers*`)`
: *NewObject* is created with *Supers* specifying the super objects (prototypes). *Supers* is a list containing super specifications. A super specification is either an object identifier or a pair *Object-NotInheritList* where *NotInheritList* specifies methods not to inherit from *Object*. *NewObject* could be an atom, variable, or compound term whose arguments are distinct variables.

The object `vehicle` is created having the proto-object `object` as super, followed by creating `moving_van` with `vehicle` as super, followed by creating `truck`.

```
| ?- object :: new(vehicle),
     vehicle :: new(moving_van),
     moving_van :: new(truck).
yes

| ?- truck :: super(X), vehicle :: sub(X).
X = moving_van ;
no
```

32.5.2 Method Additions

+*SomeObject* :: `asserta(+`*SomeMethod*`)`
+*SomeObject* :: `assertz(+`*SomeMethod*`)`
+*SomeObject* :: `assert(+`*SomeMethod*`)`
: Asserts *SomeMethod* in *SomeObject* with normal Prolog semantics.

Add some facts to `vehicle` and `truck` with initial value equal to `[]`.

```
| ?- vehicle :: assert(fuel_level([])),
     vehicle :: assert(oil_level([])),
     vehicle :: assert(location([])),
     truck :: assert(capacity([])),
     truck :: assert(total_weight([])).
yes
```

32.5.3 Parameter Passing to New Objects

When new objects are created, it is possible to pass parameters. The following example shows:

- How general methods are asserted

In the previous examples one could pass parameters to an object as follows, using the method `augment/1`.

```
| ?- vehicle :: augment({
        new_attrs(Instance, Attribute_list) :-
                self :: new(Instance),
                :: assign_list(Attribute_list, Instance) &

        assign_list([], Instance) &
        assign_list([Att|List], Instance) :-
                :: assign(Att, Instance),
                :: assign_list(List, Instance) &

        assign(P, Instance) :-
                Instance :: assert(P)
        }).
yes

% create a new 'truck'
| ?- vehicle :: new_attrs(truck, [capacity([]),total_weight([])]).
yes
```

32.6 Access Driven Programming—Daemons

Access based programming is a paradigm where certain actions are performed, or some constraints are checked, when "access operations" are invoked. Access operations for updates (i.e. assert, retract) can be redefined in an object by redefining these operations and delegating the same operation to `super`. Notice that without a delegation mechanism this would not be possible, since the *Self* would have changed. So assume that we want to print on the screen "p is augmented" whenever the fact `p(X)` is asserted in an object `foo`, we just redefine `assert/1`:

```
foo :: {
        super(object) &

        dynamic p/1 &

        p(0) &
        p(1) &

        assert(p(X)) :- !,   /* assert/1 is redefined for p(X) */
                super <: assert(p(X)),
                :display('p is augmented'), :ttynl &
```

```
assert(M) :-          /* delegating assert(_) messages */
        super <: assert(M) &
:
}.
```

32.7 Instances

Objects are relatively heavy weight. To be able to create efficiently light weight objects, we introduce the notion of *instances*. An instance is an object with restricted capability. It is created from an object that is considered its class. It gets a copy of the attributes of its class. These can be modified by `get/1` and `set/1`. An instance cannot be a class for other instances. Instances are in general very efficient, both in space and access/modification time. The attribute `'$class'/1` will store the identity of the class of the instance including parameters.

32.8 Built-In Objects and Methods

32.8.1 Universal Methods

The following methods are "universal", i.e. they are defined locally, if appropriate, for every object:

`super(`*?Object*`,`*?NotInheritList*`)`
: *Object* is a parent (a super-object) of *Self*. *NotInheritList* specifies methods of *Object* explicitly not inherited by *Self*. The definition `super(`*Object*`)` is translated to `super(`*Object*`,[])`.

`attributes(+`*Attributes*`)`
: *Attributes* is a list of compound terms specifying the local attributes of *Self* and the initial values.

32.8.2 Inlined Methods

The following methods are compiled inline i.e. calls are replaced by definitions. This implies (in the current implementation) that they have a fixed semantics an can not be redefined. There are also definitions for these methods in `object` covering the cases of unexpanded calls.

`self(`*?Self*`)`
: Unifies *Self* with "self".

`get(+`*Attribute*`)`
: Gets the attribute value(s) of the attribute specified by the head functor of *Attribute*. The value(s) are unified with the argument(s) of *Attribute*.

`set(+`*Attribute*`)`
: Sets the attribute value(s) of the attribute specified by the head functor of *Attribute*. The value(s) are taken from the argument(s) of *Attribute*.

32.8.3 The Proto-Object "object"

The proto-object `object` provides basic methods that are available to all other objects by delegation:

`super(`*?Object*`)`
: *Object* is a parent (a super-object) of *Self*. Note that any other definition of `super(`*Object*`)` are translated to the universal method `super/2`.

`sub(`*?Object*`)`
: *Object* is a child (a sub-object) of *Self*.

`self(`*?Self*`)`
: Unifies *Self* with "self". **Note:** this method is inlined when possible.

`object(`*?Object*`)`
: One of the defined objects in the system is *Object*.

`dynamic`
: *Self* is a dynamic object.

`static`
: *Self* is a static object.

`dynamic` *?Name/?Arity*
: *Name/Arity* is a dynamic method of *Self*.

`static` *?Name/?Arity*
: *Name/Arity* is a static method of *Self*.

`new(`*?Object*`)`
: Creates a new dynamic *Object*. *Self* will be the prototype of *Object*. *Object* can be a compound term, an atom, or a variable. In the last case the method generates a unique name for *Object*.

+SomeObject `::` `new(`*?NewObject*`,`*+Supers*`)`
: *NewObject* is created with *Supers* specifying the super objects (prototypes). *Supers* is a list containing super specifications. A super specification is either an object identifier or a pair *Object-NotInheritList* where *NotInheritList* specifies methods not to inherit from *Object*. *NewObject* could be an atom, variable, or compound term whose arguments are distinct variables.

`instance(`*?Instance*`)`
: Creates a new instance *Instance*. *Self* will be the class of *Instance*. *Instance* can be a compound term, an atom, or a variable. In the last case the method generates a unique name for *Instance*.

`has_instance(`*?Instance*`)`
: *Self* has the instance *Instance*.

`has_attribute(`*?AttributeSpec*`)`
: *Self* has the attribute *AttributeSpec*, locally defined or inherited. *AttributeSpec* is on the format *Name/Arity*.

`get(+`*Attribute*`)`
: Gets the attribute value(s) of the attribute specified by the head functor of *Attribute*. The value(s) are unified with the argument(s) of *Attribute*. **Note:** this method is inlined when possible.

`set(+`*Attribute*`)`
: Sets the attribute value(s) of the attribute specified by the head functor of *Attribute*. The value(s) are taken from the argument(s) of *Attribute*. **Note:** this method is inlined when possible.

`assert(+`*Fact*`)`
`assert(+`*Fact*`, -`*Ref*`)`
`asserta(+`*Fact*`)`
`asserta(+`*Fact*`, -`*Ref*`)`
`assertz(+`*Fact*`)`
`assertz(+`*Fact*`, -`*Ref*`)`
: Asserts a new *Fact* in *Self*. If *Self* is static, the name and arity of *Fact* must be declared as a dynamic method. `asserta` places *Fact* before any old facts. The other forms place it after any old facts. A pointer to the asserted fact is returned in the optional argument *Ref*, and can be used by the Prolog built-in predicates `erase/1` and `instance/2`.

`retract(+`*Fact*`)`
: Retracts a *Fact* from *Self*. If *Self* is static, the name and arity of *Fact* must be declared as a dynamic method.

`update(+`*Fact*`)`
: Replaces the first fact with the same name and arity as *Fact* in *Self* by *Fact*. If *Self* is static, the name and arity of *Fact* must be declared as a dynamic method.

`retractall(?`*Head*`)`
: Removes facts from *Self* that unify with *Head*. If *Self* is static, the name and arity of *Fact* must be declared as a dynamic method.

`abolish`
: Abolishes *Self* if dynamic.

`augment(?`*ObjectBody*`)`
`augmenta(?`*ObjectBody*`)`
`augmentz(?`*ObjectBody*`)`
: *ObjectBody*, having the form `{` *sentence-1 & ... & sentence-n* `}`, is added to *Self*. `augmenta` places the new clauses before any old clauses. The other forms place it after any old clauses.

32.8.4 The built-in object "utility"

The base object `utility` provides methods that could be used in user programs. `utility` has `object` as its super-object.

`subs(?`*Objects*`)`
: Gives a list of all the children of *Self*.

`supers(?`*Objects*`)`
: Gives a list of all parents of *Self*.

`objects(?`*Objects*`)`
: Gives a list of all objects.

`dynamic_objects(`*?Objects*`)`
: Gives a list of all dynamic objects.

`static_objects(`*?Objects*`)`
: Gives a list of all static objects.

`methods(`*?Methods*`)`
: Gives a list of all the methods of *Self*.

`dynamic_methods(`*?Methods*`)`
: Gives a list of all dynamic methods of *Self*.

`static_methods(`*?Methods*`)`
: Gives a list of all static methods of *Self*.

`descendant(`*?Object*`)`
: One of the descendants of *Self* is *Object*.

`descendant(`*?Object*`, `*?Level*`)`
: *Object* a descendant at depth *Level* of *Self*. A child of *Self* is at level 1.

`descendants(`*?Objects*`)`
: The list of all descendants of *Self* is *Objects*.

`descendants(`*?Objects*`, `*?Level*`)`
: *Objects* is the list of descendants at depth *Level* of *Self*.

`ancestor(`*?Object*`)`
: One of the ancestors of *Self* is *Object*.

`ancestor(`*?Object*`, `*?Level*`)`
: *Object* is an ancestor of *Self* at height *Level*. A super-object of *Self* has level 1.

`ancestors(`*?Object*`)`
: The list of all ancestors of *Self* is *Objects*.

`ancestors(`*?Object*`, `*?Level*`)`
: *Objects* is the list of ancestors at height *Level* of *Self*.

`restart` Removes all dynamic objects. Note that dynamic methods added to static objects are not removed.

`and_cast(+`*Objects*`, `*?Message*`)`
: Sends the same message *Message* to all objects in the list *Objects*.

`or_cast(+`*Objects*`, `*?Message*`)`
: Sends the same message *Message* to one of the objects in the list *Objects*, backtracking through the alternative objects.

32.9 Expansion to Prolog Code

As already mentioned, object definitions are expanded to Prolog clauses much as definite clause grammars. This expansion is usually transparent to the user. While debugging a Prolog Objects program, however, the expanded representation may become exposed. This section will explain in detail the source expansion, so as to give the user the possibility to relate back to the source code during a debugging session. The inheritance mechanism, based on module importation, is also described.

First of all, every statically defined object will translate to several Prolog clauses belonging to a unique *object module* with the same identity as the *object-identifier*. Object modules are significantly cheaper to create than ordinary modules, as they do not import the built-in Prolog predicates.

The module will contain predicates implementing an object declaration, the method code, imported methods and parameter transfer predicates. These predicates will be described in detail below, using the notational convention that variable names in italics are syntactic variables that will be replaced by something else in the translation process.

32.9.1 The Inheritance Mechanism

The inheritance mechanism is based on the importation mechanism of the Prolog module system. When an object is created, whether loaded from file or at runtime by `new/(1-2)`, the method predicates (i.e. predicates implementing the methods) visible in the immediate supers are collected. After subtracting from this set the method predicates which are locally defined, and those that are specified in the *don't-inherit-list* , the resulting set is made visible in the module of the inheriting object by means of importation. This implies that inherited methods are shared, expect dynamic methods.

Dynamic methods are inherited in a similar way with the big difference that they are not imported but copied. Even dynamic declarations (methods without clauses) are inherited.

Inheritance from dynamic objects differs in one aspect: Static predicates visible in a dynamic object are not imported directly from the dynamic object but from the static object from where it was imported to the dynamic object. This makes an inheriting object independent of any dynamic ancestor object after its creation.

32.9.2 Object Attributes

Attributes are based on an efficient term storage associated to modules. The attributes for an object is collected from its ancestors and itself at compile time and used for initialization at load time. The methods for accessing attributes, `get/1` and `set/1`, are inlined to primitive calls whenever possible. They should hence not be redefined.

32.9.3 Object Instances

Instances are different from other objects in that they do not inherit. Instead they share the predicate name space with its class object. They do however have their own attributes. At creation, an instance gets a copy of its class objects attributes. The reserved attribute `'$class'/1`, which is present in any object, is used for an instance to hold its class object identifier. The purpose of this is mainly to store the parameters of the class object when the instance is created.

32.9.4 The Object Declaration

The object declaration is only used by certain meta-programming operations. It consists of a fact

`'$so_type'(`*Object*`,` *Type*`).`

where *Object* is the *object-identifier*, and *Type* is either `static` or `dynamic`. If the type is `static`, the other generated predicates will be static, otherwise they will be dynamic.

32.9.5 The Method Code

Each method clause translates to a Prolog clause with two extra arguments: *Self* (a variable) and *Myself*. The latter argument is needed to cater for passing object parameters to the method body which is desribed further in next section.

The method body is translated to a Prolog-clause body as follows. The code is traversed, and the goals are transformed according to the following transformation patterns and rules. In the transformation rules, the notation *Msg*`(`*X*`,` *Y*`)` denotes the term produced by augmenting *Msg* by the two arguments *X* and *Y*:

Goal where *Goal* is a variable, is translated to `objects:call_from_body(`*Goal*`,`*Self*`,`*Myself*`,`*Src*`)` where *Src* is the source module. `objects:call_from_body/4` will meta-interpret `Goal` at runtime.

`::` *Msg* is translated to *Myself*`:`*Msg*`(`*Myself*`,`*Myself*`)` if *Msg* is a non variable. Otherwise, it is translated to `objects:call_object(`*Myself*`,` *Msg*`,` *Myself*`)`.

`<:` *Msg* is translated to *Myself*`:`*Msg*`(`*Self*`,`*Myself*`)` if *Msg* is a non variable. Otherwise, it is translated to `objects:call_object(`*Myself*`,` *Msg*`,` *Self*`)`.

`super ::` *Msg* `is translated to`
`objects:call_super_exp(`*Myself*`,`*Msg*`(`*Super*`,`*Myself*`),`*Super*`)` if *Msg* is a non variable. `call_super_exp/3` searches the supers of *Myself*. *Super* is bound to the super object where the method is found. If *Msg* is a variable, the goal is translated to `objects:call_super(`*Myself*`,`*Msg*`,`*Super*`,`*Super*`)` which expands *Msg* and performs otherwise the same actions as `call_super_exp/3`.

`super <:` *Msg*
: is translated to `objects:call_super_exp(`*Myself*`,`*Msg*`(`*Self*`,`*Myself*`),`*Super*`)` if *Msg* is a non variable. `call_super_exp/3` searches the supers of *Myself*. *Super* is bound to the super object where the method is found. If *Msg* is a variable, the goal is translated to `objects:call_super(`*Myself*`,`*Msg*`,`*Self*`,`*Super*`)` which expands *Msg* and performs otherwise the same actions as `call_super_exp/3`.

Obj `::` *Msg*
: * If *Msg* is non-variable, this is translated to *Obj*`:`*Msg*`(`*Obj*`,`*Obj*`)`.
  * Otherwise, it is translated to `objects:call_object(`*Obj*`,`*Msg*`,`*Obj*`)`.

Obj `<:` *Msg*
: * If *Msg* is non-variable, this is translated to *Obj*`:`*Msg*`(`*Self*`,`*Obj*`)`.
  * Otherwise, if *Msg* is a non-variable, it is translated to `functor(`*Obj*`,`*O*`,_)`, *O*`:`*Msg*`(`*Self*`,`*Obj*`)`.
  * Otherwise, it is translated to `objects:call_object(`*Obj*`,`*Msg*`,`*Self*`)`.

`self <:` *Msg*
`self ::` *Msg*
Msg
: are all translated like *Self* `::` *Msg*.

Module`:`*Goal*
: is translated to *Module*`:`*Goal*.

`:`*Goal*
: is translated to *Src*`:`*Goal* where *Src* is the source module.

To illustrate the expansion, consider the object `history_point` directives, all executed in the `history_point` module:

```
:-objects:create_object(history_point,
  [point-[]],
  [attributes/3,display/3,move/4,new/4,print_history/3,super/4],
  [],
  [y(0),x(0),history([])],
  tree(history_point,[tree(point,[tree(object,[])])])).

history_point:super(point, [], _, history_point).

history_point:attributes([history([])], _, _).

history_point:display(A, B, _) :-
        objects:call_super_exp(history_point, display(A,B,C), C),
        history_point:print_history(A, B, history_point).

history_point:'$so_type'(history_point, static).

history_point:move(A, B, C, _) :-
        objects:call_super_exp(history_point, move(A,B,C,E), E),
        prolog:'$get_module_data'(C, history, D),
        prolog:'$set_module_data'(C, history, [(A,B)|D]).
```

```
history_point:print_history(A, B, _) :-
        prolog:'$get_module_data'(B, history, C),
        A:format('with location history ~w~n', [C], A, A).

history_point:new(A, xy(D,E), B, _) :-
        objects:call_super_exp(history_point, new(A,xy(D,E),B,C), C),
        prolog:'$set_module_data'(A, history, [(D,E)]).
```

The directive `create_object/6` creates the object, performs the inheritance by importation, and initializes attributes. The last argument is a tree representing the ancestor hierarchy during compilation. It is used to check that the load time and compile time environments are consistent.

32.9.6 Parameter Transfer

As can be seen in the expanded methods above, the second additional argument is simply ignored if the object has no parameter. In contrast regard the following objects:

```
ellipse(RX,RY,Color) :: {
        color(Color) &
        area(A) :-
                :(A is RX*RY*3.14159265)
                        }.

circle(R,Color) :: {
        super(ellipse(R,R,Color))
                   }.

red_circle(R) :: {
        super(circle(R,red))
                  }.
```

... and their expansions:

```
ellipse(_, _, _):'$so_type'(ellipse(_,_,_), static).

ellipse(_, _, _):area(A, _, B) :-
        B:'$fix_param'(ellipse(C,D,_), B),
        user:(A is C*D*3.14159265).

ellipse(_, _, _):color(A, _, B) :-
        B:'$fix_param'(ellipse(_,_,A), B).

ellipse(_, _, _):'$fix_param'(ellipse(B,C,D), A) :-
        objects:object_class(ellipse(B,C,D), A).

circle(_, _):'$so_type'(circle(_,_), static).
```

```
circle(_, _):super(ellipse(A,A,B), [], _, circle(A,B)).

circle(_, _):'$fix_param'(circle(B,C), A) :-
        objects:object_class(circle(B,C), A).
circle(_, _):'$fix_param'(ellipse(B,B,C), A) :-
        objects:object_class(circle(B,C), A).

red_circle(_):'$so_type'(red_circle(_), static).

red_circle(_):super(circle(A,red), [], _, red_circle(A)).

red_circle(_):'$fix_param'(red_circle(B), A) :-
        objects:object_class(red_circle(B), A).
red_circle(_):'$fix_param'(circle(B,red), A) :-
        objects:object_class(red_circle(B), A).
red_circle(_):'$fix_param'(ellipse(B,B,red), A) :-
        objects:object_class(red_circle(B), A).
```

The second additional argument contains the receiver of a method call. If the method makes use of any parameter of the object where it is defined, it places a call to the reserved predicate `$fix_param/2` in the module of the receiver. The purpose of this call is to bind the parameters used in the method to appropriate values given by the receiver. The receiver may be the object where the method is defined or any of its subs. In order to service these calls, a clause of `$fix_param/2` is generated for each ancestor having parameters. Such a clause may be regarded as the collapsed chain of `super/(1-2)` definitions leading up to the ancestor.

The call `objects:object_class(`*Class*`,`*Object*`)` serves to pick up the `'$class'/1` attribute if *Object* is an instance, otherwise *Class* is unified with *Object*.

The following trace illustrates how parameters are transfered:

```
| ?- red_circle(2.5)::area(A).
1  1  Call: red_circle(2.5)::area(_A) ?
2  2  Call: ellipse(_,_,_):area(_A,red_circle(2.5),red_circle(2.5)) ?
3  3  Call: red_circle(_):$fix_param(ellipse(_B,_,_),red_circle(2.5)) ?
4  4  Call: objects:object_class(red_circle(_B),red_circle(2.5)) ?
4  4  Exit: objects:object_class(red_circle(2.5),red_circle(2.5)) ?
3  3  Exit: red_circle(_):$fix_param(ellipse(2.5,2.5,red),red_circle(2.5)) ?
5  3  Call: _A is 2.5*2.5*3.14159265 ?
5  3  Exit: 19.6349540625 is 2.5*2.5*3.14159265 ?
2  2  Exit: ellipse(_,_,_):area(19.6349540625,red_circle(2.5),red_circle(2.5)) ?
1  1  Exit: red_circle(2.5)::area(19.6349540625) ?

A = 19.6349540625 ?
```

32.10 Examples

32.10.1 Classification of Birds

This example illustrates how Prolog object can be used in classification of certain concepts. This style is common in expert system application for describing its domain.

```
animal :: {
        super(object) &
        relative_size(S) :-
                size(Obj_size),
                super(Obj_prototype),
                Obj_prototype :: size(Prototype_size),
                :(S is Obj_size/Prototype_size * 100)
                }.

bird :: {
        super(animal) &
        moving_method(fly) &
        active_at(daylight)
        }.

albatross :: {
        super(bird) &
        color(black_and_white) &
        size(115)
        }.

kiwi :: {
        super(bird) &
        moving_method(walk) &
        active_at(night) &
        size(40) &
        color(brown)
        }.

albert :: {
        super(albatross) &
        size(120)
        }.

ross :: {
        super(albatross) &
        size(40)
        }.

| ?- ross :: relative_size(R).
```

```
R = 34.78
```

32.10.2 Inheritance and Delegation

The following example illustrates a number of concepts. Firstly, how to use Prolog Objects for defining traditional classes a la Smalltalk, or other traditional object oriented languages. Secondly, how to create instances of these classes. Finally, how to access *instance variables*.

The concept of instance variables is readily available as the variables belonging to the instances created dynamically and not to the class of the instances. For example, each instance of the class `point` will have two instance variables, `x` and `y`, represented by the attributes `x/1` and `y/1`. The traditional class variables are easily available by accessing the same attributes in the associated class.

Another issue is the pattern used to create new instances. For example, to create an instance of the class `history_point`, the following code is used:

```
new(Instance, xy(IX,IY)) :-
        super <: new(Instance, xy(IX,IY)),
        Instance :: set(history([(IX,IY)])) &
```

Note that the delegation of `new/2` to `super` is necessary in order to create an object whose super is `history_point` and not `point`.

The example shows how delegation can be effective as a tool for flexible sharing of concepts in multiple inheritance. Four prototypes are defined: `point`, `history_point`, `bounded_point`, and `bh_point`. The latter is a bounded history point.

An instance of the `point` class is a point that moves in 2-D space and that can be displayed. An instance of the `history_point` class is similar to an instance of the `point` class but also keeps a history of all the moves made so far. An instance of `bounded_point` is similar to an instance of `point` but moves only in a region of the 2-D space. Finally an instance of `bh_point` inherits most of the features of a `bounded_point` and a `history_point`.

The default inheritance does not work for the methods `display/1` and `move/2` in `bh_point`. Inheritance by delegating messages to both supers of `bh_point` results in redundant actions, (moving and displaying the point twice). Selective delegation solves the problem. Taken from [Elshiewy 90].

```
point :: {
        super(object) &

        attributes([x(0),y(0)]) &

        xy(X, Y) :- get(x(X)), get(y(Y)) &

        new(Instance, xy(IX,IY)) :-
                super <: instance(Instance),
                Instance :: set(x(IX)),
```

```
                Instance :: set(y(IY)) &

        location((X,Y)) :- <: xy(X,Y) &

        move_horizontal(X) :-
                set(x(X)) &

        move_vertical(Y) :-
                set(y(Y)) &

        move(X, Y) :-
                <: move_horizontal(X),
                <: move_vertical(Y) &

        display(Terminal) :-
                <: xy(X, Y),
                Terminal :: format('point at (~d,~d)~n',[X,Y])
        }.

history_point :: {
        super(point) &

        attributes([history([])]) &

        new(Instance, xy(IX,IY)) :-
                super <: new(Instance, xy(IX,IY)),
                Instance :: set(history([(IX,IY)])) &

        move(X, Y) :-
                super <: move(X, Y),
                get(history(History)),
                set(history([(X,Y)|History])) &

        display(Terminal) :-
                super <: display(Terminal),
                <: print_history(Terminal) &

        print_history(Terminal) :-
                get(history(History)),
                Terminal :: format('with location history ~w~n',
                                   [History])
        }.

bounded_point :: {
        super(point) &

        attributes([bounds(0,0,0,0)]) &
```

```
        new(Instance, Coords, Bounds) :-
                super <: new(Instance, Coords),
                Instance :: set_bounds(Bounds) &

        set_bounds(Bounds) :-
                set(Bounds) &

        move(X, Y) :-
                <: bound_constraint(X, Y), !,
                super <: move(X, Y) &
        move(_, _) &

        bound_constraint(X, Y) :-
                get(bounds(X0, X1, Y0, Y1)),
                :(X >= X0),
                :(X =< X1),
                :(Y >= Y0),
                :(Y =< Y1) &

        display(Terminal) :-
                super <: display(Terminal),
                <: print_bounds(Terminal) &

        print_bounds(Terminal) :-
                get(bounds(X0, X1, Y0, Y1)),
                Terminal :: format('xbounds=(~d,~d), \c
                                    ybounds=(~d,~d)~n',
                                   [X0,X1,Y0,Y1])
        }.

bh_point :: {
        super(history_point) &
        super(bounded_point) &

        new(Instance, Coords, Bounds) :-
                history_point <: new(Instance, Coords),
                Instance :: set_bounds(Bounds) &

        move(X, Y) :-
                bounded_point <: bound_constraint(X, Y), !,
                history_point <: move(X, Y) &
        move(_, _) &

        display(Terminal) :-
                bounded_point <: display(Terminal),
                history_point <: print_history(Terminal)
        }.
```

```
tty :: {
        format(X, Y) :- :format(X, Y)
       }.

point at (8,12)
xbounds=(5,15), ybounds=(5,15)
with location history [(8,12),(9,11)]
```

32.10.3 Prolog++ programs

Prolog++ is a product by LPA Associates for object-oriented programming extensions of LPA Prolog. Most Prolog++ programs can be easily converted into Prolog Objects programs. The following is a translation of a program for fault diagnosis in LPA's Prolog++ manual, page 83. The program illustrates a top-down diagnosis method starting from general objects to more specific objects. The problem is fault diagnosis for car maintenance. The objects have the following structure:

```
- faults
        - electrical
        |       - lights
        |       - starting
        |               - starter_motor
        |               - sparking
        |                       - plugs
        |                       - distributer
        - fuel_system
        - mechanical
```

The general diagnosis method is defined in the object `faults`, whereas the cause-effect relationships are defined in the specific objects e.g. the object `distributor`.

This program heavily uses the `sub/1` method. We have tried to be as close as possible to the original formulation.

```
faults :: {

        super(utility) &
        dynamic(told/2) &

        /* no fault is the default */
        fault(_, _) :- :fail &

        findall :-
                <: restart,
                :: sub(Sub),
                Sub :: find(Where, Fault),
                <: print(Where, Fault),
                :fail &
```

```
        findall &

        print(Where, Fault) :-
                :writeseqnl('Location       : ', [Where]),
                :writeseqnl('Possible Fault : ', [Fault]),
                :nl &

        find(Where, Fault) :-
                self(Where),
                fault(FaultNum, Fault),
                \+ (effect(FaultNum, S),
                    contrary(S, S1),
                    exhibited(S1)
                   ),
                \+ (effect(FaultNum, SymptomNum),
                    \+ exhibited(SymptomNum)) &

        find(Where, Fault) :-
                sub(Sub),
                Sub :: find(Where, Fault) &

        exhibited(S) :-
                :: told(S, R), !,
                R = yes &
        exhibited(S) :-
                symptom(S,Text),
                (   :yesno([Text]) -> R = yes
                ;   R = no
                ),
                :: asserta(told(S,R)),
                R = yes &

        restart :-
                :: retractall(told(_,_))

        }.

electrical :: {
        super(faults)
        }.

fuel_system :: {
        super(faults)
        }.

mechanical :: {
        super(faults)
        }.
```

```
lights :: {
        super(electrical)
        }.

sparking :: {
        super(electrical)
        }.

starting :: {
        super(electrical)
        }.

starter_motor :: {
        super(electrical)
        }.

plugs :: {
        super(sparking)
        }.

engine :: {
        super(mechanical)
        }.

cylinders :: {
        super(engine)
        }.

distributor :: {
        super(sparking) &

        /* faults */
        fault('F1001', 'Condensation in distributor cap') &
        fault('F1002', 'Faulty distributor arm') &
        fault('F1003', 'Worn distributor brushes') &

        /* symptoms */
        symptom('S1001', 'Starter turns, but engine does not fire') &
        symptom('S1002', 'Engine has difficulty starting') &
        symptom('S1003', 'Engine cuts out shortly after starting') &
        symptom('S1004', 'Engine cuts out at speed') &

        /* symptoms contrary to each other */
        contrary('S1002', 'S1001') &
        contrary('S1003', 'S1001') &

        /* causal-effect relationship */
```

```
        effect('F1001', 'S1001') &
        effect('F1002', 'S1001') &
        effect('F1002', 'S1004') &
        effect('F1003', 'S1002') &
        effect('F1003', 'S1003')

        }.


yesno(Value) :- write(Value), nl, read(yes).

writeseqnl(Prompt, L) :- write(Prompt), write_seq(L).

write_seq([]).
write_seq([X|L]) :- write(X), write(' '), write_seq(L), nl.

faults :- faults :: findall.


| ?- faults.
[Starter turns, but engine does not fire]
|: yes.
Location       : distributor
Possible Fault : Condensation in distributor cap

[Engine cuts out at speed]
|: yes.
Location       : distributor
Possible Fault : Faulty distributor arm


yes
| ?- faults.
[Starter turns, but engine does not fire]
|: no.
[Engine has difficulty starting]
|: yes.
[Engine cuts out shortly after starting]
|: yes.
Location       : distributor
Possible Fault : Worn distributor brushes
```

33 Generalized Horn Clause Language

GCLA is a logic programming language (specification tool) that is based on a generalization of Prolog. This generalization is unusual in that it takes a quite different view of the meaning of a logic program—a "definitional" view rather than the traditional logical view.

GCLA has a number of noteworthy properties, for instance the possibility to do hypothetical reasoning and non-monotonic reasoning. This makes implementation of reasoning in knowledge-based systems more direct in GCLA than in many other formalisms. GCLA is also general enough to incorporate functional programming as a special case.

A GCLA program is partitioned into two parts; a declarative part and a control part that specifies the inference rules. For an introduction to GCLA and its properties see [Kreuger 91]. For methodology and programming techniques see [Aronsson 91]. Further reading can be found in [Aronsson 93] and in [Falkman & Torgersson 94].

The GCLA package is documented in its own User's Manual, located in '`library/gcla/doc/gcla.tex`' (also available as technical report SICS T91:21A). To load the package, enter the query:

```
| ?- use_module(library('gcla/gcla')).
```

34 Tcl/Tk Interface

The `tcltk` library package is a bidirectional interface to the *Tcl* (pronounced Tickle) language and the *Tk* toolkit. Tcl is an interpreted scripting language with many extension packages, in particular the graphical interface toolkit Tk.

You can read about Tcl/Tk in [Ousterhout 94] or in various articles. The articles and the Tcl/Tk system can be found by anonymous FTP, see the Release Notes for up-to-date information. This file also contains information on how to include Tcl/Tk extensions.

To load the package, enter the query:

```
| ?- use_module(library(tcltk)).
```

34.1 Prolog to Tcl

To use Tcl, you must create a Tcl interpreter object and send commands to it. The following predicate executes a single Tcl command:

```
test_command(Command, Result) :-
        tcl_new(Interp),
        tcl_eval(Interp, Command, Result),
        tcl_delete(Interp).
```

The Tcl command and its arguments is specified in *Command* (see Section 34.1.1 [Command Format], page 362), and the result will be returned as a string (list of character codes) in *Result*.

These are the predicates to use Tcl/Tk from Prolog:

`tcl_new(-`*TclInterpreter*`)`
: Creates a new interpreter, initializes it and returns a reference to it. If you don't have a standard Tcl installation you can tell Tcl where to look for the initialization files by setting the environment variable `TCL_LIBRARY`.

`tcl_delete(+`*TclInterpreter*`)`
: Deletes an interpreter and the memory used by it.

`tcl_eval(+`*TclInterpreter*`, +`*Command*`, -`*Result*`)`
: Lets *TclInterpreter* interpret the command specified by *Command*. The result will be stored as a string (list of character codes) in *Result*.

`tcl_event(+`*TclInterpreter*`, +`*Command*`, -`*Events*`)`
: This call is similar to `tcl_eval/3`. *Events* is a list of terms stored from Tcl by the `prolog_event` command (see Section 34.2 [Tcl to Prolog], page 363).

34.1.1 Command Format

A Tcl command is specified as follows:

```
Command          --> Atom              { other than [] }
                   | Number
                   | chars(PrologString)
                   | write(Term)
                   | format(Fmt,Args)
                   | dq(Command)
                   | br(Command)
                   | sqb(Command)
                   | min(Command)
                   | ListOfCommands

ListOfCommands   --> []
                   | [ListExpr]

ListExpr         --> Command
                   | Command,ListExpr
                   | Command|ListOfCommands
```

where

Atom
Number denote their printed representations.

`chars(`*PrologString*`)`
: denotes the string represented by *PrologString* (a list of character codes).

`write(`*Term*`)`
: denotes the string that is printed by the corresponding built-in predicate.

`format(`*Fmt*`,`*Args*`)`
: denotes the string that is printed by the corresponding built-in predicate.

`dq(`*Command*`)`
: denotes the string specified by *Command*, enclosed in double quotes.

`br(`*Command*`)`
: denotes the string specified by *Command*, enclosed in braces.

`sqb(`*Command*`)`
: denotes the string specified by *Command*, enclosed in square brackets.

`min(`*Command*`)`
: denotes the string specified by *Command*, immediately preceded by a hyphen.

ListOfCommands
: denotes the strings denoted by each element, separated by spaces.

Examples of command specifications and corresponding translations:

```
'set x 32'
        ==> set x 32

[set,x,br([a,b,c])]
        ==> set x {a b c}

['.panel.value_info.name',configure,min(text),br(write('$display'/1))]
        ==> .panel.value_info.name configure -text {$display/1}
```

34.2 Tcl to Prolog

There are two new Tcl commands defined in this library:

`prolog` *Goal*

Goal is a string containing the printed representation of a Prolog goal. The goal will be called in the `user` module unless it is prefixed with another module name. The call is always deterministic. The result of the call is returned as:

`"1"` if execution succeeds. The value of any of the variables in the goal that is bound to a term will be returned to Tcl in the array `prolog_variables` with the variable name as index.

The term is converted to Tcl using the same conversion as used for Tcl commands (see Section 34.1.1 [Command Format], page 362).

`"0"` if the execution fails.

Tcl exception
: if a Prolog exception is raised. The error message will be `"Prolog Exception: "` appended with a string representation of the Prolog exception.

An example:

```
test_callback(Result) :-
        tcl_new(Interp),
        tcl_eval(Interp,
                 'if {[prolog "foo(X,Y,Z)"] == 1} \\
                     {lappend res $prolog_variables(X) \\
                                  $prolog_variables(Y) \\
                                  $prolog_variables(Z)}',
                 Result),
        tcl_delete(Interp).

foo(1, bar, [a,b,c]).
```

The query

```
| ?- test_callback(Result).
```

will succeed, binding *Result* to:

```
"1 bar {a b c}"
```

`prolog_event` *Terms...*
: *Terms* are strings that contain the printed representation of Prolog terms. These are stored in a queue and retrieved as Prolog terms by `tcl_event/3` or `tk_next_event/(2,3)`.

An example:

```
test_event(Event) :-
        tcl_new(Interp),
        tcl_event(Interp, [prolog_event,dq(write(zap(42)))], Event),
        tcl_delete(Interp).
```

The query

```
| ?- test_event(Event).
```

will succeed, binding *Event* to the list `[zap(42)]`

34.3 Tk

The following example is a simple "Hello, world" program:

```
tk_hello_world :-
        tk_new([], Interp),
        tcl_eval(Interp,
                 'button .b -text "Hello" -command "destroy ."', _),
        tcl_eval(Interp, 'pack .b', _),
        tk_main_loop,
        tcl_delete(Interp).
```

The program brings up a tiny window with a button labeled "Hello" and is terminated by pressing the button.

These are the predicates to use Tk from Prolog:

`tk_new(+`*Options*`, -`*TclInterpreter*`)`
: Performs basic Tcl and Tk initialization and creates the main window of a Tk application. If you don't have a standard Tk installation you can tell Tk where to look for the initialization files by setting the environment variable `TK_LIBRARY`. *Options* is a list of optional elements according to:

  `top_level_events`
  : Event handling will be installed in the Prolog top-level loop. This means that Tk windows and the Prolog prompt can be simultaneously active.

  `name(+`*ApplicationName*`)`
  : Sets the Tk application name. The application name will be displayed in the main window and is also used for communicating between applications in Tk. Default name is an empty string.

`display(+`*Display*`)`
: Gives the name of the screen on which to create the main window. Default is normally determined by the `DISPLAY` environment variable.

`tk_main_window(+`*TclInterpreter*`, -`*TkWindow*`)`
: Gets a reference to the main window of the application using this interpreter.

`tk_make_window_exist(+`*TkWindow*`)`
: This will force a window to be mapped to the screen, i.e. it will force it to show up immediately. Normally, mapping is delayed as long as possible.

`tk_destroy_window(+`*TkWindow*`)`
: Destroys a window or widget.

`tk_num_main_windows(-`*NumberOfWindows*`)`
: Returns the number of main windows currently in use.

`tk_do_one_event`
`tk_do_one_event(+`*ListOrBitmask*`)`
: Passes control to Tk for handling a single event. This predicate may be called to give Tk time for handling the windows. You can specify what events to handle in a *List* of event flags. Alternatively a *Bitmask* as specified in the Tcl/Tk documentation can be used but should be avoided for portability between Tcl/Tk versions. The flags you can have in *List* are:

  `tk_dont_wait`
  : Don't wait for new events, process only events that are ready.

  `tk_x_events`
  `tk_window_events`
  : Process window events.

  `tk_file_events`
  : Process file events.

  `tk_timer_events`
  : Process timer events.

  `tk_idle_events`
  : Process Tk_DoWhenIdle callbacks.

  `tk_all_events`
  : Process any event.

  `tk_do_one_event/0` is equivalent to `tk_do_one_event/1` with all flags set. If the `tk_dont_wait` flag is set and there is no event to handle, the call will fail.

  It is straight-forward to define a predicate which handles all events in the queue and then returns:

```
tk_do_all_events :-
        tk_do_one_event, !,
        tk_do_all_events.
tk_do_all_events.
```

  `tk_do_one_event/(0,1)` is an interface to the C-function `TkDoOneEvent()` (or `TclDoOneEvent()` in later Tcl/Tk versions).

`tk_next_event(+`*TclInterpreter*`, -`*Event*`)`
`tk_next_event(+`*ListOrBitmask*`, +`*TclInterpreter*`, -`*Event*`)`
: Processes events until there is at least one Prolog event associated with *TclInterpreter*. *Event* is the term corresponding to the head of a queue of events (strings representing terms) stored from Tcl with the `prolog_event` command (see Section 34.2 [Tcl to Prolog], page 363). Only events associated with *TclInterpreter* are returned; other events remain in the queue. If there are no windows left the term `[]` will be returned. This predicate does not correspond directly to any Tcl/Tk C function.

`tk_main_loop`
: Passes control to Tk until all windows are gone.

There are basically two alternatives for invoking Prolog actions on user events. The first is to directly call Prolog by means of the Tcl command `prolog`. Tcl/Tk must then be repeatedly invoked, either by calling `tk_main_loop/0` or using the option `top_level_events`, possibly in conjunction with calling `tk_do_one_event/(0,1)` in lengthy computations.

The second alternative is using the Tcl command `prolog_event`. The user program then, by calling `tk_next_event/(2,3)`, passes control to Tcl/Tk until one or more invocations of `prolog_event` has occurred. The Prolog event determines the Prolog action to take. This approach has the advantage that context variables might be passed around in the event loop.

35 The Gauge Profiling Tool

The Gauge library package is a graphical interface to the Sicstus built-in predicates `profile_data/4` and `profile_reset/1`. See Section 7.15 [Profiling], page 121, for more information about execution profiling. The interface is based on Tcl/Tk (see Chapter 34 [TclTk], page 361). Gauge loads Tcl/Tk if it is not previously loaded.

To use the gauge package, enter the query

```
| ?- use_module(library(gauge)).
```

`view(`*:Spec*`)`

Creates a graphical user interface for viewing the profile data for the predicates covered by *Spec*, which has the same form as for `spy/1` (see Section 6.3 [Spy-Point], page 62). For example, the call `view([user:_,m2:_])`, will bring up the graphical user interface on the predicates contained in the modules `user` and `m2`. When the display first comes up it is blank except for the control panel.

The graphical interface is pretty much self explanatory. The on-line help function gives all necessary information.

36 I/O on Lists of Character Codes

This package defines I/O predicates that read from, or write to, a list of character codes (a string). There are also predicates to open a stream referring to a list of character codes. The stream may be used with general Stream I/O predicates.

To load the package, enter the query

```
| ?- use_module(library(charsio)).
```

`format_to_chars(+`*Format*`, +`*Arguments*`, -`*Chars*`)`
`format_to_chars(+`*Format*`, +`*Arguments*`, ?`*S0*`, ?`*S*`)`
: Prints *Arguments* into a list of character codes using `format/3` (see Section 7.1.3 [Term I/O], page 80). *Chars* is unified with the list, alternatively *S0* and *S* are unified with the head and tail of the list, respectively.

`write_to_chars(+`*Term*`, -`*Chars*`)`
`write_to_chars(+`*Term*`, ?`*S0*`, ?`*S*`)`
: A specialized `format_to_chars/(3-4)`. Writes *Term* into a list of character codes using `write/2` (see Section 7.1.3 [Term I/O], page 80). *Chars* is unified with the list, alternatively *S0* and *S* are unified with the head and tail of the list, respectively.

`atom_to_chars(+`*Atom*`, -`*Chars*`)`
`atom_to_chars(+`*Atom*`, ?`*S0*`, ?`*S*`)`
: A specialized `format_to_chars/(3-4)`. Converts *Atom* to the list of characters comprising its name. *Chars* is unified with the list, alternatively *S0* and *S* are unified with the head and tail of the list, respectively.

`number_to_chars(+`*Number*`, -`*Chars*`)`
`number_to_chars(+`*Number*`, ?`*S0*`, ?`*S*`)`
: A specialized `format_to_chars/(3-4)`. Converts *Number* to the list of characters comprising its name. *Chars* is unified with the list, alternatively *S0* and *S* are unified with the head and tail of the list, respectively.

`read_from_chars(+`*Chars*`, -`*Term*`)`
: Reads *Term* from *Chars* using `read/2`. The *Chars* must, as usual, be terminated by a *full-stop*, i.e. a ., possibly followed by *layout-text*.

`open_chars_stream(+`*Chars*`, -`*Stream*`)`
: *Stream* is opened as an input stream to an existing list of character codes. The stream may be read with the *Stream I/O* predicates and must be closed using `close/1`. The list is copied to an internal buffer when the stream is opened and must therefore be a ground list of character codes at that point.

`with_output_to_chars(+`*Goal*`, -`*Chars*`)`
`with_output_to_chars(+`*Goal*`, ?`*S0*`, ?`*S*`)`

`with_output_to_chars(+`*Goal*`, -`*Stream*`, `*?S0*`, `*?S*`)`

Goal is called with the `current_output` stream set to a new stream. This stream writes to an internal buffer which is, after the successful execution of *Goal*, converted to a list of character codes. *Chars* is unified with the list, alternatively *S0* and *S* are unified with the head and tail of the list, respectively. `with_output_to_chars/4` also passes the stream in the *Stream* argument. It can be used only by *Goal* for writing.

37 Jasper

The Jasper library module provides functionality for accessing the Java VM from Prolog. The current functionality is limited to calling methods using the standard foreign language interface. See Chapter 9 [Mixing Java and Prolog], page 167. However, this will most likely be extended in the future.

The library does not yet define any predicates. However, it must be loaded to make sure that the JVM (the Java Virtual Machine) is loaded correctly and that the interface code has access to all relevant references to the JVM. It is loaded by the query

```
| ?- use_module(library(jasper)).
```

This will dynamically load the JVM into memory and initialize the JNI Invocation API.

There is also an directory of examples of how to use Jasper: `library('jasper/examples')`.

38 Glue Code Generator

This chapter describes utilities useful for generating glue code for the Foreign Language Interface when building statically linked Runtime Systems or Development Systems (see Section 8.2.5 [Interface Predicates], page 133).

This library module can be loaded by the query

```
| ?- use_module(library(flinkage)).
```

and refers to the following shared object files or DLLs:

runtime kernel
: The SICStus runtime kernel, usually '`$SP_PATH/../libsprt37.so`' under UNIX, or '`%SP_PATH%\..\sprt37.dll`' under Windows, and

development kernel
: The SICStus development kernel, usually '`$SP_PATH/../libspds37.so`' under UNIX, or '`%SP_PATH%\..\spds37.dll`' under Windows.

The following predicate can be used to generate a glue code file '`flinkage.c`' file out of any old-style `foreign_file/2` and `foreign/(2,3)` declarations found in a set of Prolog files for use in a statically linked executable. **Note:** this is provided for backwards compatibility only. We recommend using foreign resources instead, which can be dynamically linked:

`generate_flinkage(+`*Files*`)`
: *Files* are loaded as by `use_module/1`. Calls to `load_foreign_files/2` are intercepted by means of the goal expansion mechanism. The file '`flinkage.c`' is generated out of the declarations found. Finally, compile and link the '`flinkage.c`' file with the user code, libraries and the runtime and optionally the development kernel.

The following predicates, available in Development Systems only, provide alternative methods to building executables or linked foreign resources compared to the methods described in Section 8.8.3 [Creating the Executable], page 154, Section 8.9 [Development Systems], page 156 and Section 8.2.7 [Creating the Linked Foreign Resource], page 135. If the default method doesn't work as expected for some reason or you wish to use an Integrated Development Environment etc., the following outlines the necessary steps. To build an executable:

First, decide which foreign resources are to be statically linked and create a *resource table file* as follows:

`prepare_resource_table(+`*ResourceNames*`,+`*CFile*`)`
: where *+ResourceNames* is a list of the resource names and *+CFile* is the file where to write the generated code.

The next step is to create linked foreign resources from the foreign resources which are to be statically linked. Produce, for each foreign resource, a *glue code file* using:

`prepare_foreign_resource(+`*ResourceName*`,+`*SourceFile*`,+`*CFile*`)`
: where *ResourceName* is the name of the resource, *SourceFile* is the Prolog source file containing the resource conversion declarations and *CFile* is the file where to write the generated code.

Finally, compile and link the glue code files with the user code, libraries and the runtime and optionally the development kernel. Further information (platform specific) is provided with the distribution.

To build a dynamic linked foreign resource:

Produce a glue code file using `prepare_foreign_resource/3`. Check the provided platform specific notes for compiler options needed and compile the glue code file and the user code files. Likewise check the notes on how a dynamic linked foreign resource is implemented on this platform and link the object files to an appropriate object.

39 Timeout Predicate

The `timeout` package, which may be loaded by the query

```
| ?- use_module(library(timeout)).
```

contains the predicate:

`time_out(`*:Goal*`, +`*Time*`, ?`*Result*`)`
: The *Goal* is executed as if by `call/1`. If computing any solution takes more than *Time* milliseconds, process virtual time, the goal will fail and *Result* is unified with the atom `time_out`. If the goal succeeds within the specified time *Result* is unified with the atom `success`. *Time* must be a number between (not including) 0 and 2147483647.

`time_out/3` is implemented by raising and handling `time_out` exceptions, so any exception handler in the scope of *Goal* must be prepared to pass on `time_out` exceptions. The following incorrect example shows what can happen otherwise:

```
| ?- time_out(on_exception(Q,(repeat,false),true), 1000, Res).
Q = time_out,
Res = success
```

40 Cross Reference Producer

This package provides a cross reference producer is a useful tool for debugging and program analysis. It can be used to produce the call graph of a program, and to produce a list of predicates that are used but not defined or vice versa. The graph processing utilities described in previous chapters may be used to analyze the call graphs.

To load the package, enter the query

```
| ?- use_module(library(xref)).
```

`call_graph(`*:Files*`, -`*Graph*`)`
: *Graph* is the call graph (see Chapter 23 [UGraphs], page 221) of the predicates in *Files*. The vertices of *Graph* are terms of the following types:

  `def`
  : There is an edge from `def` to every predicate that is explicitly defined, declared, imported, or implicitly defined in a `foreign/(2-3)` fact.

  `use`
  : There is an edge from `use` to every predicate that is used in a directive, exported, or declared as `public`. If a "hook" predicate such as `user:portray/1` is defined, there is an edge from `use` to it too.

  Module`:`*Name*`/`*Arity*
  : Denotes a predicate occurring in one of the *Files*. There is an edge from predicate P to predicate Q if P calls Q and Q is not a built-in predicate.

`xref(`*:Files*`)`
: *Files* is a single file or a list of files, as accepted by `consult/1` and friends. The predicate prints on the current output stream a list of predicates which are reachable from `use` in the call graph but not defined, followed by a list of predicates which are defined but not reachable from `use`.

There may be references not found by `xref` because of the dynamic nature of the Prolog language. Note that `xref` does not consider a predicate used unless it is reachable from a directive, an export list, or a `public` declaration. In non-module files, it is therefore recommended that the entry point predicates be declared as `public`. Predicates reached by meta-calls only may also have to be declared `public` to be considered used.

If a module file (file containing a module declaration) is encountered, the module specified is used for that file *and* subsequent files up to the next module file. This enables packages consisting of several files, which are to be loaded in the same module, to be correctly treated. The type-in module is the default module.

Summary of Built-In Predicates

`!` Commits to any choices taken in the current predicate.

`(+`*P*`,+`*Q*`)` *P* and *Q*.

`(`*Head* `-->` *Body*`)`
Not a built-in predicate; reserved syntax for grammar rules.

`(+`*P* `-> +`*Q* `; +`*R*`)`
If *P* then *Q* else *R*, using first solution of *P* only.

`(+`*P* `-> +`*Q*`)`
If *P* then *Q* else false, using first solution of *P* only.

`[]`
`[:`*File*`|+`*Files*`]`
Updates the program with interpreted clauses from *File* and *Files*.

`(:-` *Command*`)`
Not a built-in predicate; reserved syntax for commands and declarations.

`(?-` *Query*`)`
Not a built-in predicate; reserved syntax for queries.

`(`*Head* `:-` *Body*`)`
Not a built-in predicate; reserved syntax for clauses.

`(+`*P*`;+`*Q*`)` *P* or *Q*.

`?`*X* `=` `?`*Y* The terms *X* and *Y* are unified.

`+`*Term* `=..` *?List*
`?`*Term* `=..` `+`*List*
The functor and arguments of the term *Term* comprise the list *List*.

`+`*X* `=:=` `+`*Y*
X is numerically equal to *Y*.

?Term1 `==` *?Term2*
The terms *Term1* and *Term2* are strictly identical.

`+`*X* `=\=` `+`*Y*
X is not numerically equal to *Y*.

`+`*X* `=<` `+`*Y* *X* is less than or equal to *Y*.

`+`*X* `>` `+`*Y* *X* is greater than *Y*.

`+`*X* `>=` `+`*Y* *X* is greater than or equal to *Y*.

`?`*X* `^` `:`*P* Executes the procedure call *P*.

`\+` `+`*P* Goal *P* is not provable.

?Term1 `\==` *?Term2*
The terms *Term1* and *Term2* are not strictly identical.

`+`*X* `<` `+`*Y* *X* is less than *Y*.

?Term1 `@=<` *?Term2*
: The term *Term1* precedes or is identical to the term *Term2* in the standard order.

?Term1 `@>` *?Term2*
: The term *Term1* follows the term *Term2* in the standard order.

?Term1 `@>=` *?Term2*
: The term *Term1* follows or is identical to the term *Term2* in the standard order.

?Term1 `@<` *?Term2*
: The term *Term1* precedes the term *Term2* in the standard order.

`?=(`*?X*`,`*?Y*`)`
: *X* and *Y* are either syntactically identical or syntactically non-unifiable.

`abolish(`*:Preds*`)`
: Makes the predicate(s) specified by *Preds* undefined.

`abolish(`*:Atom*`,`*+Arity*`)`
: Makes the predicate specified by *Atom*/*Arity* undefined.

`abort` Aborts execution of the current directive (returns to C in recursive calls to Prolog from C).

`absolute_file_name(`*+RelativeName*`,` *?AbsoluteName*`)`
: *AbsoluteName* is the full pathname of *RelativeName*.

`arg(`*+ArgNo*`,`*+Term*`,` *?Arg*`)`
: The argument *ArgNo* of the term *Term* is *Arg*.

`assert(`*:Clause*`)`
`assert(`*:Clause*`,`*-Ref*`)`
: Asserts clause *Clause* with unique identifier *Ref*.

`asserta(`*:Clause*`)`
`asserta(`*:Clause*`,`*-Ref*`)`
: Asserts *Clause* as first clause with unique identifier *Ref*.

`assertz(`*:Clause*`)`
`assertz(`*:Clause*`,`*-Ref*`)`
: Asserts *Clause* as last clause with unique identifier *Ref*.

`at_end_of_line`
`at_end_of_line(`*Stream*`)`
: The end of stream or end of line has been reached for *Stream* or from the current input stream.

`at_end_of_stream`
`at_end_of_stream(`*Stream*`)`
: The end of stream has been reached for *Stream* or from the current input stream.

`atom(`*?X*`)` *X* is currently instantiated to an atom.

`atom_chars(+`*Atom*`,`*?CharList*`)`
`atom_chars(`*?Atom*`,+`*CharList*`)`
: The name of the atom *Atom* is the list of characters *CharList*.

`atomic(`*?X*`)`
: *X* is currently instantiated to an atom or a number.

`bagof(`*?Template*`,`*:Goal*`,`*?Bag*`)`
: *Bag* is the bag of instances of *Template* such that *Goal* is satisfied (not just provable).

`block` *Specs*
: Not a built-in predicate; block declaration.

`bb_delete(+`*Key*`,`*?Term*`)`
: Delete from the blackboard *Term* stored under *Key*.

`bb_get(+`*Key*`,`*?Term*`)`
: Get from the blackboard *Term* stored under *Key*.

`bb_put(+`*Key*`,+`*Term*`)`
: Store the term *Term* under *Key* on the blackboard.

`bb_update(`*:Key*`, `*?OldTerm*`, `*?NewTerm*`)`
: Replace the term *OldTerm* by the term *NewTerm* under *Key* on the blackboard.

`break` Invokes a recursive top-level. (not available in Runtime Systems).

`'C'(`*?S1*`,`*?Terminal*`,`*?S2*`)`
: *Grammar rules. S1* is connected by the terminal *Terminal* to *S2*.

`call(`*:Term*`)`
`(`*Module*`::`*Term*`)`
: Executes the procedure call *Term* in *Module*.

`call_cleanup(`*:Goal*`,`*:Cleanup*`)`
: Executes the procedure call *Goal*. When *Goal* succeeds deterministically, is cut, fails, or raises an exception, *Cleanup* is executed.

`call_residue(`*:Goal*`,`*?Residue*`)`
: Executes the procedure call *Goal*. Any floundered goals and the variables they are blocked on occur as *VarSet-Goal* pairs in *Residue*.

`callable(`*?X*`)`
: *X* is currently instantiated to a compound term or an atom.

`character_count(+`*Stream*`,`*?Count*`)`
: *Count* characters have been read from or written to the stream *Stream*.

`clause(`*:Head*`,`*?Body*`)`
`clause(`*:Head*`,`*?Body*`,`*?Ref*`)`
`clause(`*?Head*`,`*?Body*`,+`*Ref*`)`
: There is an interpreted clause whose head is *Head*, whose body is *Body*, and whose unique identifier is *Ref*.

`close(+`*Stream*`)`
: Closes stream *Stream*.

`compare(`*?Op*`,`*?Term1*`,`*?Term2*`)`
: *Op* is the result of comparing the terms *Term1* and *Term2*.

`compile(`*:Files*`)`
: Compiles in-core the clauses in text file(s) *Files*.

`compound(`*?X*`)`
: *X* is currently instantiated to a term of arity > 0.

`consult(`*:Files*`)`
: Updates the program with interpreted clauses from file(s) *Files*.

`copy_term(`*?Term*`,`*?CopyOfTerm*`)`
: *CopyOfTerm* is an independent copy of *Term*.

`create_mutable(+`*Datum*`,-`*Mutable*`)`
: *Mutable* is a new mutable term with current value *Datum*.

`current_atom(`*?Atom*`)`
: One of the currently defined atoms is *Atom*.

`current_input(`*?Stream*`)`
: *Stream* is the current input stream.

`current_key(`*?KeyName*`,`*?KeyTerm*`)`
: There is a recorded item in the internal database whose key is *KeyTerm*, the name of which is *KeyName*.

`current_module(`*?Module*`)`
: *Module* is a module currently in the system.

`current_module(`*?Module*`,`*?File*`)`
: *Module* is a module currently in the system, loaded from *File*.

`current_op(`*?Precedence*`,`*?Type*`,`*?Op*`)`
: Atom *Op* is an operator type *Type* precedence *Precedence*.

`current_output(`*?Stream*`)`
: *Stream* is the current output stream.

`current_predicate(`*?Name*`,`*:Head*`)`
`current_predicate(`*?Name*`,`*-Head*`)`
: A user defined or library predicate is named *Name*, most general goal *Head*.

`current_stream(`*?AbsFileName*`,`*?Mode*`,`*?Stream*`)`
: There is a stream *Stream* associated with the file *AbsFileName* and opened in mode *Mode*.

`debug`
: Switches on debugging in *leap* mode (not available in Runtime Systems).

`debugging`
: Displays debugging status information (not available in Runtime Systems).

`dif(`*?X*`,`*?Y*`)`
: The terms *X* and *Y* are different.

`display(`*?Term*`)`
: Displays the term *Term* on the standard output stream.

`dynamic` *Specs*
: Not a built-in predicate; dynamic declaration.

`ensure_loaded(`*:Files*`)`
: Compiles or loads the file(s) *Files* if need be.

`erase(+`*Ref*`)`
: Erases the clause or record whose unique identifier is *Ref*.

`error_exception(+`*Exception*`)`
`user:error_exception(+`*Exception*`)`
: *A hook predicate*, *Exception* is an exception that traps to the debugger if it is switched on.

`expand_term(+`*Term1*`,` *?Term2*`)`
: The term *Term1* is a shorthand which expands to the term *Term2*.

`fail`
`false`
: False.

`fcompile(`*:Files*`)`
: Compiles file-to-file the clauses in text file(s) *Files* (not available in Runtime Systems).

`file_search_path(`*Alias*`,` *?Expansion*`)`
`user:file_search_path(`*Alias*`,` *?Expansion*`)`
: *A hook predicate*, telling how to expand *Alias*`(`*File*`)` file names.

`fileerrors`
: Enables reporting of file errors.

`findall(`*?Template*`,` *:Goal*`,` *?Bag*`)`
`findall(`*?Template*`,` *:Goal*`,` *?Bag*`,` *?Remainder*`)`
: A prefix of *Bag* is the list of instances of *Template* such that *Goal* is provable. The rest of *Bag* is *Remainder* or the empty list.

`float(`*?X*`)`
: *X* is currently instantiated to a float.

`flush_output`
`flush_output(+`*Stream*`)`
: Flushes the buffers associated with *Stream*.

`foreign(+`*CFunctionName*`,+`*Predicate*`)`
`foreign(+`*CFunctionName*`,+`*Language*`,+`*Predicate*`)`
: *Hook predicates*, tell Prolog how to define *Predicate* to invoke *CFunctionName*.

`foreign_file(+`*ObjectFile*`,+`*Functions*`)`
: *A hook predicate*, tells Prolog that foreign functions *Functions* are in file *ObjectFile*. Obsolete, use `foreign_resource/2` instead.

`foreign_resource(+`*ResourceName*`,+`*Functions*`)`
: *A hook predicate*, tells Prolog that foreign functions *Functions* are in resource *ResourceName*.

`format(+`*Format*`,+`*Arguments*`)`
`format(+`*Stream*`,+`*Format*`,+`*Arguments*`)`
: Writes *Arguments* according to *Format* on the stream *Stream* or on the current output stream.

`freeze(?`*Var*`,:`*Goal*`)`
: Blocks *Goal* until `nonvar(`*Var*`)` holds.

`frozen(-`*Var*`,?`*Goal*`)`
: The goal *Goal* is blocked on the variable *Var*.

`functor(+`*Term*`,?`*Name*`,?`*Arity*`)`
`functor(?`*Term*`,+`*Name*`,+`*Arity*`)`
: The principal functor of the term *Term* has name *Name* and arity *Arity*.

`garbage_collect`
: Performs a garbage collection of the global stack.

`garbage_collect_atoms`
: Performs a garbage collection of the atoms.

`gc` : Enables garbage collection of the global stack.

`get(?`*C*`)`
`get(+`*Stream*`,?`*C*`)`
: The next printing character from the stream *Stream* or from the current input stream is *C*.

`get0(?`*C*`)`
`get0(+`*Stream*`,?`*C*`)`
: The next character from the stream *Stream* or from the current input stream is *C*.

`get_mutable(?`*Datum*`,+`*Mutable*`)`
: The current value of the mutable term *Mutable* is *Datum*.

`goal_expansion(+`*Goal*`,+`*Module*`,-`*NewGoal*`)`
`user:goal_expansion(+`*Goal*`,+`*Module*`,-`*NewGoal*`)`
: *A hook predicate.* Defines a transformation from *Goal* in module *Module* to *NewGoal*.

`ground(?`*X*`)`
: *X* is currently free of unbound variables.

`halt` : Halts Prolog. (returns to C in recursive calls to Prolog from C).

`halt(`*Code*`)`
: Halts Prolog immediately, returning *Code*.

`help` : *Hookable*, prints a help message (not available in Runtime Systems).

`if(+`*P*`,+`*Q*`,+`*R*`)`
: If *P* then *Q* else *R*, exploring all solutions of *P*.

`incore(+`*Term*`)`
: Executes the procedure call *Term*.

`initialization`
: Executes a list of user defined goals (not available in Runtime Systems).

`initialization(`*:Goal*`)`
: Adds *Goal* to the list of goals executed by `initialization/0` (not available in Runtime Systems).

`instance(+`*Ref*`,` *?Term*`)`
: *Term* is a most general instance of the record or clause uniquely identified by *Ref*.

`integer(`*?X*`)`
: *X* is currently instantiated to an integer.

?Y `is +`*X* : *Y* is the value of the arithmetic expression *X*.

`is_mutable(`*?X*`)`
: *X* is currently instantiated to a mutable term.

`keysort(+`*List1*`,` *?List2*`)`
: The list *List1* sorted by key yields *List2*.

`leash(+`*Mode*`)`
: Sets leashing mode to *Mode* (not available in Runtime Systems).

`length(`*?List*`,` *?Length*`)`
: The length of list *List* is *Length*.

`library_directory(`*?Directory*`)`
`user:library_directory(`*?Directory*`)`
: *A hook predicate*, telling how to expand `library(`*File*`)` file names.

`line_count(+`*Stream*`,` *?N*`)`
: *N* is the number of lines read/written on stream *Stream*.

`line_position(+`*Stream*`,` *?N*`)`
: *N* is the number of characters read/written on the current line of *Stream*.

`link_foreign_resource(+`*Resource*`,+`*SourceFile*`,+`*Option*`,+`*CFiles*`,+`*ObjectFiles*`,+`*Libraries*`)`
: Builds a linked foreign resource (not available in Runtime Systems).

`listing`
`listing(`*:Specs*`)`
: Lists the interpreted predicate(s) specified by *Specs* or all interpreted predicates in the type-in module. Any variables in the listed clauses are internally bound to ground terms before printing. If this causes any blocked goals to be executed, the behavior is undefined.

`load(`*:Files*`)`
: Loads object file(s) *Files*.

`load_files(`*:Files*`)`
`load_files(`*:Files*`, +`*Options*`)`
: Loads source or object file(s) *Files* obeying *Options*.

`load_foreign_files(`*:ObjectFiles*`,+`*Libraries*`)`
: *Hookable*, links object files *ObjectFiles* into Prolog. Obsolete, use `link_foreign_resource/6` and `load_foreign_resource/1` instead.

`load_foreign_resource(`*:Resource*`)`
: Loads foreign resource *Resource* into Prolog.

`unload_foreign_resource(`*:Resource*`)`
: Unloads foreign resource *Resource* from Prolog.

`meta_predicate` *Specs*
: Not a built-in predicate; meta-predicate declaration.

`method_expansion(+`*Method1*`,`*?Object*`,`*?Method2*`)`
`user:method_expansion(+`*Method1*`,`*?Object*`,`*?Method2*`)`
: *A hook predicate*, defines transformations on Prolog Object methods.

`mode` *Specs*
: Not a built-in predicate; mode declaration.

`module(+`*Module*`)`
: Sets the type-in module to *Module*; see Section 4.1 [Basic Concepts], page 47.

`module(+`*Module*`, +`*ExportList*`)`
`module(+`*Module*`, +`*ExportList*`, +`*Options*`)`
: Not a built-in predicate; module declaration.

`multifile` *Specs*
: Not a built-in predicate; multifile declaration.

`muse_flag(`*?FlagName*`,`*?Value*`)`
: *Value* is the current value of the Muse flag *FlagName*.

`muse_flag(+`*FlagName*`,`*?OldValue*`,`*?NewValue*`)`
: *OldValue* and *NewValue* are the old and new values of the Muse flag *FlagName*.

`muse_flags`
: Display all Muse flag settings.

`muse_sync`
: Synchronize for being leftmost in Muse.

`muse_trace(`*:Goal*`)`
`muse_trace(`*:Goal*`,+`*FilePrefix*`)`
: Generate a Muse trace file for *Goal*.

`name(+`*Const*`,`*?CharList*`)`
`name(`*?Const*`,+`*CharList*`)`
: The name of atom or number *Const* is the string *CharList*.

`nl`
`nl(+`*Stream*`)`
: Outputs a new line on stream *Stream* or on the current output stream.

`nodebug` Switches off debugging (not available in Runtime Systems).

`nofileerrors`
: Disables reporting of file errors.

`nogc` Disables garbage collection of the global stack.

`nonvar(`*?X*`)`
: *X* is a non-variable.

`nospy` *:Spec*
: Removes spy-points from the predicate(s) specified by *Spec* (not available in Runtime Systems).

`nospyall` Removes all spy-points (not available in Runtime Systems).

`notrace`
`nozip` Switches off debugging (not available in Runtime Systems).

`number(`*?X*`)`
: *X* is currently instantiated to a number.

`number_chars(+`*Number*`,`*?CharList*`)`
`number_chars(`*?Number*`,+`*CharList*`)`
: The name of the number *Number* is the list of characters *CharList*.

`numbervars(`*?Term*`,+`*N*`,`*?M*`)`
: Number the variables in the term *Term* from *N* to *M*-1.

`on_exception(`*?Pattern*`,`*:ProtectedGoal*`,`*:Handler*`)`
: Executes the procedure call *ProtectedGoal*. If during the execution `raise_exception(`*Exception*`)` is called, and *Exception* matches *Pattern*, the execution of *ProtectedGoal* abort, *Pattern* is unified with a copy of *Exception* and *Handler* is called.

`op(+`*Precedence*`,+`*Type*`,+`*Name*`)`
: Makes atom(s) *Name* an operator of type *Type* precedence *Precedence*.

`open(+`*FileName*`,+`*Mode*`,-`*Stream*`)`
`open(+`*FileName*`,+`*Mode*`,-`*Stream*`,+`*Options*`)` *[ISO]*
: Opens file *FileName* in mode *Mode* with options *Options* as stream *Stream*.

`open_null_stream(-`*Stream*`)`
: Opens an output stream to the null device.

`otherwise`
: True.

`parallel`
`parallel` *Specs*
: Not a built-in predicate; parallel declaration.

`peek_char(`*?N*`)`
`peek_char(+`*Stream*`,`*?N*`)`
: *N* is the character code of the next character peeked at from *Stream* or from the current input stream.

`phrase(`*:Phrase*`,`*?List*`)`
`phrase(`*:Phrase*`,`*?List*`,`*?Remainder*`)`
: *Grammar rules.* The list *List* can be parsed as a phrase of type *Phrase*. The rest of the list is *Remainder* or empty.

`portray(+`*Term*`)`
`user:portray(+`*Term*`)`
: *A hook predicate*, tells `print/1` what to do.

`portray_clause(+`*Clause*`)`
`portray_clause(+`*Stream*`,+`*Clause*`)`
: Pretty prints *Clause* on the stream *Stream* or on the current output stream.

`portray_message(+`*Severity*`,+`*Message*`)`
`user:portray_message(+`*Severity*`,+`*Message*`)`
: *A hook predicate*, tells `print_message/2` what to do.

`predicate_property(:`*Head*`,?`*Prop*`)`
`predicate_property(-`*Head*`,?`*Prop*`)`
: *Head* is the most general goal of a currently defined predicate that has the property *Prop*.

`print(?`*Term*`)`
`print(+`*Stream*`,?`*Term*`)`
: *Hookable*, portrays or else writes the term *Term* on the stream *Stream* or on the current output stream.

`print_message(+`*Severity*`,+`*Message*`)`
: *Hookable*, portrays or else writes *Message* of a given *Severity* on the standard error stream.

`profile_data(:`*Files*`,?`*Selection*`,?`*Resolution*`,-`*Data*`)`
: *Data* is the profiling data collected from the instrumented predicates defined in the file(s) *Files* with selection and resolution *Selection* and *Resolution* respectively (not available in Runtime Systems nor in Muse).

`profile_reset(:`*Files*`)`
: The profiling counters for the instrumented predicates in the file(s) *Files* are zeroed (not available in Runtime Systems nor in Muse).

`prolog_flag(?`*FlagName*`,?`*Value*`)`
: *Value* is the current value of *FlagName*.

`prolog_flag(+`*FlagName*`,?`*OldValue*`,?`*NewValue*`)`
: *OldValue* and *NewValue* are the old and new values of *FlagName*.

`prolog_load_context(?`*Key*`,?`*Value*`)`
: *Value* is the value of the compilation/loading context variable identified by *Key*.

`prompt(?`*Old*`,?`*New*`)`
: Changes the prompt from *Old* to *New*.

`public` *Specs*
: Not a built-in predicate; public declaration.

`put(+`*C*`)`
`put(+`*Stream*`,+`*C*`)`
: The next character sent to the stream *Stream* or to the current output stream is *C*.

`raise_exception(+`*Exception*`)`
: Causes abortion of a part of the execution tree scoped by the closest enclosing `on_exception/3` invocation with its first argument matching *Exception*.

`read(?`*Term*`)`
`read(+`*Stream*`,?`*Term*`)`
: Reads the term *Term* from the stream *Stream* or from the current input stream.

`read_term(?`*Term*`,+`*Options*`)`
`read_term(+`*Stream*`,?`*Term*`,+`*Options*`)`
: Reads the term *Term* from the stream *Stream* or from the current input stream with extra *Options*.

`reconsult(:`*Files*`)`
: Updates the program with interpreted clauses from file(s) *Files*.

`recorda(+`*Key*`,?`*Term*`,-`*Ref*`)`
: Makes the term *Term* the first record under key *Key* with unique identifier *Ref*.

`recorded(?`*Key*`,?`*Term*`,?`*Ref*`)`
: The term *Term* is currently recorded under key *Key* with unique identifier *Ref*.

`recordz(+`*Key*`,?`*Term*`,-`*Ref*`)`
: Makes the term *Term* the last record under key *Key* with unique identifier *Ref*.

`reinitialise`
: Initializes Prolog (returns to C in recursive calls to Prolog from C).

`repeat` Succeeds repeatedly.

`require(:`*PredSpecs*`)`
: Tries to locate and load library files that export the specified predicates. Creates index files if necessary (not available in Runtime Systems).

`restore(+`*File*`)`
: Restores the state saved in file *File*.

`retract(:`*Clause*`)`
: Erases repeatedly the next interpreted clause of form *Clause*.

`retractall(:`*Head*`)`
: Erases all clauses whose head matches *Head*.

`save_program(+`*File*`)`
`save_program(+`*File*`, :`*Goal*`)`
: Saves the current state of the Prolog data base in file *File*. Upon restore, *Goal* is executed.

`see(+`*File*`)`
: Makes file *File* the current input stream.

`seeing(`*?File*`)`
: The current input stream is named *File*.

`seek(+`*Stream*`,+`*Offset*`,+`*Method*`,-`*NewLocation*`)`
: Sets the stream *Stream* to the byte offset *Offset* relative to *Method*, and *NewLocation* is the new byte offset from the beginning of the file after the operation.

`seen`
: Closes the current input stream.

`sequential`
`sequential` *Specs*
: Not a built-in predicate; sequential declaration.

`set_input(+`*Stream*`)`
: Sets the current input stream to *Stream*.

`set_output(+`*Stream*`)`
: Sets the current output stream to *Stream*.

`set_stream_position(+`*Stream*`,+`*Position*`)`
: *Position* is a term representing a new position of *Stream*, which is then set to the new position.

`setof(`*?Template*`,`*:Goal*`,`*?Set*`)`
: *Set* is the set of instances of *Template* such that *Goal* is satisfied (not just provable).

`simple(`*?X*`)`
: *X* is currently uninstantiated or atomic.

`skip(+`*C*`)`
`skip(+`*Stream*`,+`*C*`)`
: Skips characters from *Stream* or from the current input stream until after character *C*.

`skip_line`
`skip_line(+`*Stream*`)`
: Skips characters from *Stream* or from the current input stream until the next `LFD`.

`sort(+`*List1*`,`*?List2*`)`
: The list *List1* sorted into order yields *List2*.

`source_file(`*?File*`)`
`source_file(`*:Pred*`,`*?File*`)`
`source_file(-`*Pred*`,`*?File*`)`
: The predicate *Pred* is defined in the file *File*.

`spy` *:Spec*
: Sets spy-points on the predicate(s) specified by *Spec* (not available in Runtime Systems).

`spypoint_condition(`*:Goal*`,`*?Port*`,+`*Test*`)`
: Sets conditional spy-point on the predicate for *Goal* (not available in Runtime Systems).

`statistics`
: Outputs various execution statistics.

`statistics(`*?Key*`,`*?Value*`)`
: The execution statistics key *Key* has value *Value*.

`stream_code(+`*Stream*`,`*?StreamCode*`)`
`stream_code(`*?Stream*`,+`*StreamCode*`)`
: *StreamCode* is an integer representing a pointer to the internal representation of *Stream*.

`stream_interrupt(`*?Stream*`,`*?OldHandler*`,`*?NewHandler*`)`
: Sets/reads the interrupt handler connected to the stream *Stream* (not available in Muse).

`stream_position(+`*Stream*`,`*?Position*`)`
: *Position* is a term representing the current position of *Stream*.

`stream_select(+`*Streams*`,+`*TimeOut*`,-`*ReadStreams*`)`
: Returns a list *ReadStreams* containing streams with pending characters. Only the streams in the list *Streams* are checked. *TimeOut* specifies a timeout on the form `off` or `Sec:MicroSec` (not available in Muse).

`tab(+`*N*`)`
`tab(+`*Stream*`,+`*N*`)`
: Outputs *N* spaces to the stream *Stream* or to the current output stream.

`tell(+`*File*`)`
: Makes file *File* the current output stream.

`telling(`*?File*`)`
: The current output stream is named *File*.

`term_expansion(+`*Term1*`,`*?TermOrTerms*`)`
`term_expansion(+`*Term1*`,+`*Layout1*`,`*?TermOrTerms*`,`*?Layout2*`)`
`user:term_expansion(+`*Term1*`,`*?TermOrTerms*`)`
`user:term_expansion(+`*Term1*`,+`*Layout1*`,`*?TermOrTerms*`,`*?Layout2*`)`
: *A hook predicate*, tells `expand_term/2` what to do.

`told` Closes the current output stream.

`trace` Switches on debugging in *creep* mode (not available in Runtime Systems).

`true` Succeeds.

`ttyflush` Flushes the standard output stream buffer.

`ttyget(`*?C*`)`
: The next printing character input from the standard input stream is *C*.

`ttyget0(`*?C*`)`
: The next character input from the standard input stream is *C*.

`ttynl` Outputs a new line on the standard output stream.

`ttyput(+`*C*`)`
: The next character output to the standard output stream is *C*.

`ttyskip(+`*C*`)`
: Skips characters from the standard input stream until after character *C*.

`ttytab(+`*N*`)`
: Outputs *N* spaces to the standard output stream.

`unknown(`*?OldState*`,`*?NewState*`)`
: Changes action on undefined predicates from *OldState* to *NewState* (not available in Runtime Systems).

`unknown_predicate_handler(+`*Goal*`,+`*Module*`,-`*NewGoal*`)`
`user:unknown_predicate_handler(+`*Goal*`,+`*Module*`,-`*NewGoal*`)`
: *A hook predicate.* Defines an alternative goal to be called in place of a call to an unknown predicate.

`update_mutable(+`*Datum*`,+`*Mutable*`)`
: Updates the current value of the mutable term *Mutable* to become *Datum*.

`use_module(:`*Files*`)`
: Loads the module file(s) *Files* if necessary and import all public predicates.

`use_module(:`*File*`,+`*Imports*`)`
: Loads the module file *File* if necessary and imports the predicates in *Imports*.

`use_module(+`*Module*`,?`*File*`,+`*Imports*`)`
`use_module(?`*Module*`,:`*File*`,+`*Imports*`)`
: Equivalent to `use_module/2` plus unifies *Module* to the module defined by the file.

`user_help`
`user:user_help`
: *A hook predicate*, tells `help/0` what to do.

`var(?`*X*`)` *X* is currently uninstantiated.

`version` Displays introductory and/or system identification messages (not available in Runtime Systems).

`version(+`*Message*`)`
: Adds the atom *Message* to the list of introductory messages (not available in Runtime Systems).

`volatile` *Specs*
: Not a built-in predicate; volatile declaration.

`when(+`*Condition*`,:`*Goal*`)`
: Blocks *Goal* until the *Condition* is true.

`write(+`*Term*`)`
`write(+`*Stream*`,+`*Term*`)`
: Writes the term *Term* on the stream *Stream* or on the current output stream.

`write_canonical(+`*Term*`)`
`write_canonical(+`*Stream*`,+`*Term*`)`
: Writes *Term* on the stream *Stream* or on the current output stream so that it may be read back.

`write_term(+`*Term*`,+`*Options*`)`
`write_term(+`*Stream*`,+`*Term*`,+`*Options*`)`
: Writes the term *Term* on the stream *Stream* or on the current output stream with extra *Options*.

`writeq(+`*Term*`)`
`writeq(+`*Stream*`,+`*Term*`)`
: Writes the term *Term* on the stream *Stream* or on the current output stream, quoting names where necessary.

`zip`
: Switches on debugging in *zip* mode (not available in Runtime Systems).

Full Prolog Syntax

A Prolog program consists of a sequence of *sentences* or lists of sentences. Each sentence is a Prolog *term*. How terms are interpreted as sentences is defined below (see [Sentence], page 396). Note that a term representing a sentence may be written in any of its equivalent syntactic forms. For example, the 2-ary functor ':-' could be written in standard prefix notation instead of as the usual infix operator.

Terms are written as sequences of *tokens*. Tokens are sequences of characters which are treated as separate symbols. Tokens include the symbols for variables, constants and functors, as well as punctuation characters such as brackets and commas.

We define below how lists of tokens are interpreted as terms (see [Term Token], page 397). Each list of tokens which is read in (for interpretation as a term or sentence) has to be terminated by a full-stop token. Two tokens must be separated by a layout-text token if they could otherwise be interpreted as a single token. Layout-text tokens are ignored when interpreting the token list as a term, and may appear at any point in the token list.

We define below defines how tokens are represented as strings of characters (see [Token String], page 398). But we start by describing the notation used in the formal definition of Prolog syntax (see [Syntax Notation], page 395).

Notation

1. Syntactic categories (or *non-terminals*) are written thus: *item*. Depending on the section, a category may represent a class of either terms, token lists, or character strings.
2. A syntactic rule takes the general form

   C `-->` *F1* `|` *F2* `|` *F3*

   which states that an entity of category *C* may take any of the alternative forms *F1*, *F2*, *F3*, etc.
3. Certain definitions and restrictions are given in ordinary English, enclosed in `{ }` brackets.
4. A category written as *C...* denotes a sequence of one or more *C*s.
5. A category written as *?C* denotes an optional *C*. Therefore *?C...* denotes a sequence of zero or more *C*s.
6. A few syntactic categories have names with arguments, and rules in which they appear may contain meta-variables looking thus: *X*. The meaning of such rules should be clear from analogy with the definite clause grammars (see Section 7.1.2 [Definite], page 76).
7. In the section describing the syntax of terms and tokens (see [Term Token], page 397) particular tokens of the category name are written thus: *name*, while tokens which are individual punctuation characters are written literally.

Syntax of Sentences as Terms

sentence `-->` *module* `:` *sentence*
| *list*
`{` where *list* is a list of *sentence* `}`
| *clause*
| *directive*
| *grammar-rule*

clause `-->` *non-unit-clause* `|` *unit-clause*

directive `-->` *command* `|` *query*

non-unit-clause `-->` *head* `:-` *body*

unit-clause `-->` *head*
`{` where *head* is not otherwise a *sentence* `}`

command `-->` `:-` *body*

query `-->` `?-` *body*

head `-->` *module* `:` *head*
| *goal*
`{` where *goal* is not a *variable* `}`

body `-->` *module* `:` *body*
| *body* `->` *body* `;` *body*
| *body* `->` *body*
| `\+` *body*
| *body* `;` *body*
| *body* `,` *body*
| *goal*

goal `-->` *term*
`{` where *term* is not otherwise a *body* `}`

grammar-rule `-->` *gr-head* `-->` *gr-body*

gr-head `-->` *module* `:` *gr-head*
| *gr-head* `,` *terminals*
| *non-terminal*
`{` where *non-terminal* is not a *variable* `}`

gr-body `-->` *module* `:` *gr-body*
| *gr-body* `->` *gr-body* `;` *gr-body*
| *gr-body* `->` *gr-body*
| `\+` *gr-body*
| *gr-body* `;` *gr-body*

```
                    |  gr-body , gr-body
                    |  non-terminal
                    |  terminals
                    |  gr-condition

non-terminal     --> term
                         { where term is not otherwise a gr-body }

terminals        --> list | string

gr-condition     --> ! | { body }

module           --> atom
```

Syntax of Terms as Tokens

```
term-read-in     --> subterm(1200) full-stop

subterm(N)       --> term(M)
                         { where M is less than or equal to N }

term(N)          --> op(N,fx) subterm(N-1)
                         { except in the case of a number }
                         { if subterm starts with a (,
                           op must be followed by layout-text }
                    |  op(N,fy) subterm(N)
                         { if subterm starts with a (,
                           op must be followed by layout-text }
                    |  subterm(N-1) op(N,xfx) subterm(N-1)
                    |  subterm(N-1) op(N,xfy) subterm(N)
                    |  subterm(N) op(N,yfx) subterm(N-1)
                    |  subterm(N-1) op(N,xf)
                    |  subterm(N) op(N,yf)

term(1000)       --> subterm(999) , subterm(1000)

term(0)          --> functor ( arguments )
                         { provided there is no layout-text between
                           the functor and the ( }
                    |  ( subterm(1200) )
                    |  { subterm(1200) }
                    |  list
                    |  string
                    |  constant
                    |  variable

op(N,T)          --> name
```

```
                                { where name has been declared as an
                                  operator of type T and precedence N }

arguments          --> subterm(999)
                     |  subterm(999) , arguments

list               --> []
                     |  [ listexpr ]

listexpr           --> subterm(999)
                     |  subterm(999) , listexpr
                     |  subterm(999) | subterm(999)

constant           --> atom | number

number             --> unsigned-number
                     |  sign unsigned-number
                     |  sign inf
                     |  sign nan

unsigned-number    --> natural-number | unsigned-float

atom               --> name

functor            --> name
```

Syntax of Tokens as Character Strings

By default, SICStus Prolog uses the ISO 8859/1 character set standard, but will alternatively support the EUC (Extended UNIX Code) standard. This is governed by the value of the environment variable `SP_CTYPE` (see Section 1.1 [Start], page 7).

The character categories used below are defined as follows in the two standards:

layout-char
: In ISO 8859/1, these are character codes 0..32 and 127..159. In EUC, these are character codes 0..32 and 127. The common subset includes characters such as `TAB`, `LFD`, and `SPC`.

small-letter
: In ISO 8859/1, these are character codes 97..122, 223..246, and 248..255. In EUC, these are character codes 97..122 and 128..255. The common subset includes the letters `a` through `z`.

capital-letter
: In ISO 8859/1, these are character codes 65..90, 192..214, and 216..222. In EUC, these are character codes 65..90. The common subset is the letters `A` through `Z`.

digit In both standards, these are character codes 48..57, i.e. the digits `0` through `9`.

symbol-char
In ISO 8859/1, these are character codes 35, 36, 38, 42, 43, 45..47, 58, 60..64, 92, 94, 96, 126, 160..191, 215, and 247. In EUC, these are character codes 35, 36, 38, 42, 43, 45..47, 58, 60..64, 92, 94, 96, and 126. The common subset is
`+-*/\^<>=‘~:.?@#$&`.

solo-char In both standards, these are character codes 33 and 59 i.e. the characters `!` and `;`.

punctuation-char
In both standards, these are character codes 37, 40, 41, 44, 91, 93, and 123..125, i.e. the characters `%()`,`[]{|}`.

quote-char
In both standards, these are character codes 34 and 39 i.e. the characters `"` and `’`.

underline In both standards, this is character code 95 i.e. the character `_`.

token `-->` *name*
`|` *natural-number*
`|` *unsigned-float*
`|` *variable*
`|` *string*
`|` *punctuation-char*
`|` *layout-text*
`|` *full-stop*

name `-->` *quoted-name*
`|` *word*
`|` *symbol*
`|` *solo-char*
`|` `[` *?layout-text* `]`
`|` `{` *?layout-text* `}`

quoted-name `-->` `’` *?quoted-item...* `’`

quoted-item `-->` *char* `{` other than `’` or `\` `}`
`|` `’’`
`|` `\` *escape-sequence*

word `-->` *small-letter* *?alpha...*

symbol `-->` *symbol-char...*
`{` except in the case of a *full-stop*
or where the first 2 chars are `/*` `}`

natural-number `-->` *digit...*
`|` *base* `’` *alpha...*
`{` where each *alpha* must be less than the *base*,
treating a,b,... and A,B,... as 10,11,... `}`

```
                    |  0 ' char-item
                           { yielding the character code for char }

char-item          --> char   { other than \ }
                    |  \ escape-sequence

base               --> digit...   { in the range [2..36] }

unsigned-float     --> simple-float
                    |  simple-float exp exponent

simple-float       --> digit... . digit...

exp                 --> e  |  E

exponent            --> digit... | sign digit...

sign                --> - | +

variable            --> underline ?alpha...
                    |  capital-letter ?alpha...

string              --> " ?string-item... "

string-item        --> char   { other than " or \ }
                    |  ""
                    |  \ escape-sequence

layout-text              --> layout-text-item...

layout-text-item         --> layout-char | comment

comment             --> /* ?char... */
                           { where ?char... must not contain */ }
                    |  % ?char... LFD
                           { where ?char... must not contain LFD }

full-stop          --> .
                           { the following token, if any, must be layout-text}

char                --> { any character, i.e. }
                         layout-char
                    |  alpha
                    |  symbol-char
                    |  solo-char
                    |  punctuation-char
                    |  quote-char
```

```
alpha              --> capital-letter | small-letter | digit | underline

escape-sequence    --> b          { backspace, character code 8 }
                     | t            { horizontal tab, character code 9 }
                     | n            { newline, character code 10 }
                     | v            { vertical tab, character code 11 }
                     | f            { form feed, character code 12 }
                     | r            { carriage return, character code 13 }
                     | e            { escape, character code 27 }
                     | d            { delete, character code 127 }
                     | a            { alarm, character code 7 }
                     | x alpha alpha
                         {treating a,b,... and A,B,... as 10,11,... }
                                    { in the range [0..15], hex character code }
                     | digit ?digit ?digit
                                    { in the range [0..7], octal character code }
                     | ^ ?          { delete, character code 127 }
                     | ^ capital-letter
                     | ^ small-letter
                                    { the control character alpha mod 32 }
                     | c ?layout-char...  { ignored }
                     | layout-char   { ignored }
                     | char      { other than the above, represents itself }
```

Escape Sequences

A backslash occurring inside integers in '`0`'' notation or inside quoted atoms or strings has special meaning, and indicates the start of an escape sequence. Character escaping can be turned off for compatibility with old code. The following escape sequences exist:

`\b`	backspace (character code 8)
`\t`	horizontal tab (character code 9)
`\n`	newline (character code 10)
`\v`	vertical tab (character code 11)
`\f`	form feed (character code 12)
`\r`	carriage return (character code 13)
`\e`	escape (character code 27)
`\d`	
`\^?`	delete (character code 127)
`\a`	alarm (character code 7)
`\x`*CD*	the character code *CD* (two hexadecimal digits)
`\`*octal*	the character code *octal* base 8, where *octal* is up to 3 octal digits

`\^`*char* the character code *char* `mod 32`, where *char* is a letter.

`\`*layout-char*
A single layout character, for example a newline, is ignored.

`\c` All characters up to, but not including, the next non-layout character are ignored.

`\`*other* A character not mentioned in this table stands for itself. For example, '`\\`' inserts a single backslash and '`\'`' inserts a single quote.

Notes

1. The expression of precedence 1000 (i.e. belonging to syntactic category *term(1000)*) which is written
   X`,`Y
   denotes the term `','(`X`,`Y`)` in standard syntax.
2. The parenthesized expression (belonging to syntactic category *term(0)*)
   `(`X`)`
   denotes simply the term X.
3. The curly-bracketed expression (belonging to syntactic category *term(0)*)
   `{`X`}`
   denotes the term `{}(`X`)` in standard syntax.
4. Note that, for example, `-3` denotes a number whereas `-(3)` denotes a compound term which has the 1-ary functor `-` as its principal functor.
5. The character `"` within a string must be written duplicated. Similarly for the character `'` within a quoted atom.
6. Backslashes in strings, quoted atoms, and integers written in '`0'`' notation denote escape sequences.
7. A name token declared to be a prefix operator will be treated as an atom only if no *term-read-in* can be read by treating it as a prefix operator.
8. A name token declared to be both an infix and a postfix operator will be treated as a postfix operator only if no *term-read-in* can be read by treating it as an infix operator.

Standard Operators

```
:- op( 1200, xfx, [ :-, --> ]).
:- op( 1200,  fx, [ :-, ?- ]).
:- op( 1150,  fx, [ mode, public, dynamic, volatile,
                    multifile, block, meta_predicate,
                    parallel, sequential ]).
:- op( 1100, xfy, [ ; ]).
:- op( 1050, xfy, [ -> ]).
:- op( 1000, xfy, [ ',' ]).
:- op(  900,  fy, [ \+, spy, nospy ]).
:- op(  700, xfx, [ =, is, =.., ==, \==, @<, @>, @=<, @>=,
                                  =:=, =\=, <, >, =<, >= ]).
:- op(  550, xfy, [ : ]).
:- op(  500, yfx, [ +, -, #, /\, \/ ]).
:- op(  500,  fx, [ +, - ]).
:- op(  400, yfx, [ *, /, //, <<, >> ]).
:- op(  300, xfx, [ mod ]).
:- op(  200, xfy, [ ^ ]).
```

References

[Aggoun & Beldiceanu 92]
A. Aggoun and N. Beldiceanu, *Extending CHIP in order to Solve Complex Scheduling and Placement Problems*, Mathl. Comput. Modelling, vol. 17, no. 7, pp. 57–73, Pergamon Press Ltd., 1993.

[Ali & Karlsson 92]
K.A.M. Ali and R. Karlsson, *Scheduling Speculative work in MUSE and Performance Results*, International Journal of Parallel Programming, Vol 21, No.6, December 1992.

[Aronsson 91]
M. Aronsson, *Methodology and Programming Techniques in GCLAII.* In *Proceedings of the second Workshop on Extensions of Logic Programming (ELP)*, by Eriksson, Hallnas, and Schroeder-Heister (eds.), held at SICS, Stockholm, Sweden 1991, to be published in Lecture Notes in Artificial Intelligence, Springer-Verlag. Also available as research report SICS R92:05.

[Aronsson 93]
M. Aronsson, *Implementational Issues in GCLA: A-Sufficiency and the Definiens operation*, in Proceedings of the third Workshop on Extensions of Logic Programming (ELP), LNAI 660, Springer Verlag, 1993.

[Beldiceanu & Contejean 94]
N. Beldiceanu and E. Contejean, *Introducing Global Constraints in CHIP*, Mathl. Comput. Modelling, vol. 20, no. 12, pp. 97–123, Pergamon Press Ltd., 1994.

[Bryant 86]
R.E. Bryant, *Graph-Based Logarithms for Boolean Function Manipulation*, IEEE Trans. on Computers, August, 1986.

[Carlsson 90]
M. Carlsson, *Design and Implementation of an OR-Parallel Prolog Engine*, SICS Dissertation Series 02, 1990.

[Carreiro & Gelernter 89a]
N. Carreiro and D. Gelernter, *Linda in Context*, *Comm. of the ACM*, 32(4) 1989.

[Carreiro & Gelernter 89b]
N. Carreiro and D. Gelernter, *How to Write Parallel Programs: A Guide to the Perplexed*, *ACM Computing Surveys*, September 1989.

[Clocksin & Mellish 81]
W.F. Clocksin and C.S. Mellish, *Programming in Prolog*, Springer-Verlag, 1981.

[Colmerauer 75]
A. Colmerauer, *Les Grammaires de Metamorphos*, Technical Report, Groupe d'Intelligence Artificielle, Marseille-Luminy, November, 1975.

[Colmerauer 90]
Colmerauer A.: An Introduction to Prolog III, Communications of the ACM, 33(7), 69-90, 1990.

[Conery 83]
J. Conery, *The AND/OR Process Model for Parallel Interpretation of Logic Programs*, PhD thesis, University of California at Irvine, 1983.

[Diaz & Codognet 93]
D. Diaz and P. Codognet, *A Minimal Extension of the WAM for clp(FD)*, Proceedings of the International Conference on Logic Programming, MIT Press, 1993.

[Elshiewy 90]
N.A. Elshiewy, *Robust Coordinated Reactive Computing in Sandra*, SICS Dissertation Series 03, 1990.

[Falkman & Torgersson 94]
G. Falkman and O. Torgersson, *Programming Methodologies in GCLA*, in Proceedings of the fourth Workshop on Extensions of Logic Programming (ELP), LNAI 798, Springer Verlag, 1994.

[Gorlick & Kesselman 87]
M.M. Gorlick and C.F. Kesselman, *Timing Prolog Programs Without Clocks*, Proc. Symposium on Logic Programming, pp. 426–432, IEEE Computer Society, 1987.

[Hausman 90]
B. Hausman, *Pruning and Speculative Work in OR-Parallel PROLOG*, PhD thesis, Department of Telecommunication and Computer Systems, The Royal Institute of Technology, Stockholm, 1990.

[Heintze et al. 87]
N. Heintze, J. Jaffar, S. Michaylov, P. Stuckey, R. Yap, The CLP(R) Programmers Manual, Monash University, Clayton, Victoria, Australia, Department of Computer Science, 1987.

[Hermenegildo and Greene 90]
M.V. Hermenegildo and K.J. Greene, *&-Prolog and its performance: Exploiting Independent And-Parallelism*, Proceedings of the Seventh ICLP, Jerusalem, MIT Press, 1990.

[Holzbaur 90]
C. Holzbaur, *Specification of Constraint Based Inference Mechanism through Extended Unification*, dissertation, Dept. of Medical Cybernetics & AI, University of Vienna, 1990.

[Holzbaur 92]
C. Holzbaur, *A High-Level Approach to the Realization of CLP Languages*, Proceedings of the JICSLP92 Post-Conference Workshop on Constraint Logic Programming Systems, Washington D.C., 1992.

[Holzbaur 92]
C. Holzbaur, *Metastructures vs. Attributed Variables in the Context of Extensible Unification*, in M. Bruynooghe & M. Wirsing (eds.), Programming Language Implementation and Logic Programming, Springer-Verlag, LNCS 631, pp. 260-268, 1992.

[Holzbaur 94]
C. Holzbaur, *A Specialized, Incremental Solved Form Algorithm for Systems of Linear Inequalities*, Austrian Research Institute for Artificial Intelligence, Vienna, TR-94-07, 1994.

[Jaffar & Michaylov 87]
J. Jaffar, S. Michaylov, *Methodology and Implementation of a CLP System*, in J.L. Lassez (ed.), Logic Programming - Proceedings of the 4th International Conference - Volume 1, MIT Press, Cambridge, MA, 1987.

[Karlsson 92]
R. Karlsson, *A High Performance OR-parallel Prolog System*, PhD thesis, Department of Telecommunication and Computer Systems, The Royal Institute of Technology, Stockholm, 1992.

[Kowalski 74]
R.A. Kowalski, *Logic for Problem Solving*, DCL Memo 75, Dept of Artificial Intelligence, University of Edinburgh, March, 1974.

[Kowalski 79]
R.A. Kowalski, *Artificial Intelligence: Logic for Problem Solving.* North Holland, 1979.

[Kreuger 91]
P. Kreuger, *GCLAII—A definitional approach to control.* In *Proceedings of the second Workshop on Extensions of Logic Programming (ELP)*, by Eriksson, Hallnas, and Schroeder-Heister (eds.), held at SICS, Stockholm, Sweden 1991, to be published in Lecture Notes in Artificial Intelligence, Springer-Verlag. Also available as research report SICS R92:09.

[Lusk et al. 90]
E. Lusk, D.H.D. Warren, S. Haridi et al., *The Aurora Or-Parallel System*, New Generation Computing vol. 7 nos. 2–3 pp. 243–271, 1990.

[McCabe 92]
F. McCabe, *Logic and Objects*, Prentice Hall, 1992.

[O'Keefe 90]
R.A. O'Keefe, *The Craft of Prolog*, MIT Press, 1990.

[Ousterhout 94]
John K. Ousterhout, *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[Pereira & Warren 80]
F.C.N. Pereira and D.H.D. Warren, *Definite clause grammars for language analysis—a survey of the formalism and a comparison with augmented transition networks* in Artificial Intelligence 13:231-278, 1980.

[Regin 94] J.-C. Regin, *A filtering algorithm for constraints of difference in CSPs*, Proc. of the Twelfth National Conference on Artificial Intelligence (AAAI-94), pp. 362–367, 1994

[Robinson 65]
J.A. Robinson, *A Machine-Oriented Logic Based on the Resolution Principle*, Journal of the ACM 12:23-44, January 1965.

[Roussel 75]
P. Roussel, *Prolog : Manuel de Reference et d'Utilisation*, Groupe d'Intelligence Artificielle, Marseille-Luminy, 1975.

[Saraswat 90]
V. Saraswat, *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie-Mellon University, 1990.

[Sterling & Shapiro 86]
L. Sterling and E. Shapiro, *The Art of Prolog.* The MIT Press, Cambridge MA, 1986.

[Van Hentenryck 89]
P. Van Hentenryck, *Constraint Satisfaction in Logic Programming*, Logic Programming Series, The MIT Press, 1989.

[Van Hentenryck et al. 92]
P. Van Hentenryck, V. Saraswat and Y. Deville, *Constraint processing in cc(FD)*, unpublished manuscript, 1992.

[Van Hentenryck & Deville 91]
P. Van Hentenryck and Y. Deville, *The Cardinality Operator: a new logical connective and its application to constraint logic programming*, In: Eighth International Conference on Logic Programming, 1991.

[Van Hentenryck et al. 95]
P. Van Hentenryck, V. Saraswat and Y. Deville, *Design, implementation and evaluation of the constraint language cc(FD).* In A. Podelski, ed., *Constraints: Basics and Trends*, volume 910 of *Lecture Notes in Computer Science.* Springer-Verlag, 1995.

[Warren 77]
D.H.D. Warren, *Applied Logic—Its Use and Implementation as a Programming Tool*, PhD thesis, Edinburgh University, 1977. Available as Technical Note 290, SRI International.

[Warren 83]
D.H.D. Warren, *An Abstract Prolog Instruction Set*, Technical Note 309, SRI International, 1983.

[Wirth 76] N. Wirth, *Algorithms + Data Structures = Programs*, Prentice-Hall, 1976.

Predicate Index

!

!/0, cut 36, 99

#

/2, bitwise exclusive or 94
/2, boolean eor 241
#/\ /2 277
#= /2 275
#=> /2 277
#=< /2 275
#> /2 275
#>= /2 275
#\ /1 276
#\ /2 277
#\/ /2 277
#\= /2 275
#< /2 275
#<= /2 277
#<=> /2 ..... 275, 277

*

* /2, boolean and 241
* /2, multiplication 94

,

,/2, conjunction 99

-

- /1, negation 94
- /2, subtraction 94
--> /2, grammar rule 76
-> /2 ;/2, if then else 99
-> /2, if then 99

.

. /2, consult 74
. /2, identity 94

/

/ /2, floating division 94
// /2, integer division 94
/\ /2, bitwise conjunction 94

:

:- /1, command 13
:- /2, clause 31
:/2 100
::/1 (message sending) 332
::/2 (message sending) 332

;

;/2, disjunction 99

=

= /2, unification 124
=../2, univ 111
=:= /2, arithmetic equal 96
=:= /2, boolean equal 241
== /2, equality of terms 97
=\= /2, arithmetic not equal 96
=\= /2, boolean not equal 241
=< /2, arithmetic less or equal 96
=< /2, boolean less or equal 241

?

?- /1, query 12
?= /2, terms identical or cannot unify 98

[

[]/0, consult 74

@

@=< /2, term less or equal 98
@> /2, term greater than 98
@>= /2, term greater or equal 98
@< /2, term less than 98

{

{}/1 248

~

~ /1, boolean not 241

+

+ /1, identity 94
+ /2, addition 94
+ /2, boolean ior 241

>

> /2, arithmetic greater than 96
> /2, boolean greater 241
>= /2, arithmetic greater or equal 97
>= /2, boolean greater or equal 241
>> /2, right shift 94

^

^ /2, boolean existential quantifier 241
^ /2, existential quantifier 119

\

\ /1, bitwise negation 94
\/ /2, bitwise disjunction 94
\== /2, inequality of terms 97
\+ /1, not provable 99

<

< /2, arithmetic less than 96
< /2, boolean less 241
<:/1 (message delegation) 332
<:/2 (message delegation) 332
<< /2, left shift 94

A

abolish/1 115
abolish/2 115
abort (debugger command) 67
abort/0 ..... 17, 125
abs/1, absolute value 95
absolute_file_name/2 89
acos/1, function 95
acosh/1, function 96
acot/1, function 95
acot2/2, function 95
acoth/1, function 96
acyclic_term/1 206
add_edges/3 ..... 222, 226
add_to_heap/4 199
add_vertices/3 ..... 221, 225
agc_margin (prolog flag) 105
all_different/1 278
all_distinct/1 278
ancestors (debugger command) 67
append/3 201
aref/3 189
arefa/3 189
arefl/3 189
arg/3 111
argv (prolog flag) 105
array_to_list/2 189
aset/4 190
asin/1, function 95
asinh/1, function 95
assert/1 114
assert/2 114
asserta/1 114
asserta/2 114
assertz/1 114
assertz/2 114
assignment/2 278
assoc_to_list/2 191
at_end_of_line/0 92
at_end_of_line/1 92
at_end_of_stream/0 91
at_end_of_stream/1 91
atan/1, function 95
atan2/2, function 95
atanh/1, function 96
atom/1 110
atom_chars/2 112
atom_garbage_collection (statistics/2 option) 109
atom_to_chars/2 369
atom_to_chars/3 369
atomic/1 110
atoms (statistics/2 option) 109
attribute/1 (declaration) 193
attribute_goal/2 194

B

backtrace (debugger command) 67
backtracks (profile_data/4 option) 122
bagof/3 119
bagof_rd_noblock/3 234
bb_delete/2 117
bb_get/2 116
bb_inf/3 250
bb_inf/5 251
bb_put/2 116
bb_update/3 117
block/1 (declaration) 55
blocked goals (debugger command) 67
break (debugger command) 68
break/0 ..... 16, 125

C

C/3 ... 80
call/1 ... 100
call_cleanup/2 ... 101
call_graph/2 ... 377
call_residue/2 ... 120, 261
callable/1 ... 110
calls (profile_data/4 option) ... 122
card/2, boolean cardinality ... 241
ceiling/1, function ... 95
character_count/2 ... 91
character_escapes (prolog flag) ... 106
choice (statistics/2 option) ... 109
choice_points (profile_data/4 option) ... 122
CHOICESTKSIZE (environment) ... 10
chr_debug/0 ... 317
chr_debugging/0 ... 317
chr_leash/1 ... 318
chr_nodebug/0 ... 317
chr_nospy/1 ... 320
chr_notrace/0 ... 317
chr_spy/1 ... 319
chr_trace/0 ... 317
circuit/1 ... 278
circuit/2 ... 278
clause/2 ... 114
clause/3 ... 114
clique/3 ... 223
close/1 ... 89
close_client/0 ... 233
clpfd:dispatch_global/4 ... 285
clpfd:full_answer/0 ... 283
coloring/3 ... 223
colouring/3 ... 223
command (debugger command) ... 68
compare/3 ... 98
compile-buffer (emacs command) ... 20
compile-file (emacs command) ... 20
compile-predicate (emacs command) ... 20
compile-region (emacs command) ... 20
compile/1 ... 52, 74
compiling (prolog flag) ... 105
complement/2 ... 222
compose/3 ... 222
compound/1 ... 110
consistency_error/4 (error class) ... 103
consult-buffer (emacs command) ... 20
consult-file (emacs command) ... 20
consult-predicate (emacs command) ... 20
consult-region (emacs command) ... 20
consult/1 ... 52, 74
context_error/3 (error class) ... 103
copy_term/2 ... 112
core (statistics/2 option) ... 109
cos/1, function ... 95
cosh/1, function ... 95
cot/1, function ... 95
coth/1, function ... 95
count/4 ... 277
create_mutable/2 ... 113
creep (debugger command) ... 65
cumulative/4 ... 279
current_atom/1 ... 104
current_constraint/2 ... 313
current_db/4 ... 237
current_handler/2 ... 313
current_host/1 ... 230
current_input/1 ... 90
current_key/2 ... 116
current_module/1 ... 105
current_module/2 ... 105
current_op/3 ... 124
current_output/1 ... 90
current_predicate/2 ... 104
current_stream/3 ... 90
cyclic_term/1 ... 206

D

datime/1 ... 215
db_buffering/2 ... 238
db_buffering/3 ... 238
db_canonical/2 ... 238
db_canonical/3 ... 238
db_close/0 ... 237
db_close/1 ... 237
db_erase/1 ... 237
db_erase/2 ... 237
db_fetch/2 ... 237
db_fetch/3 ... 237
db_findall/2 ... 237
db_findall/3 ... 237
db_open/3 ... 236
db_open/4 ... 236
db_store/2 ... 237
db_store/3 ... 237
debug/0 ... 61, 120
debugger_print_options (prolog flag) ... 108
debugging (debugger command) ... 67
debugging (prolog flag) ... 105
debugging/0 ... 62, 121

deep_fails (profile_data/4 option) 122
del_assoc/4 192
del_edges/3 ..... 222, 226
del_max_assoc/4 192
del_min_assoc/4 192
del_vertices/3 ..... 221, 226
delete/3 201
delete_file/1 215
delete_file/2 215
delete_from_heap/4 200
dif/2 120
directory_files/2 215
display (debugger command) 66
display/1 82
domain/3 276
domain_error/4 (error class) 103
dump/3 251
dynamic/1 (declaration) 54

E

edges/2 ..... 221, 225
element/3 277
empty_assoc/1 191
empty_fdset/1 287
empty_heap/1 199
empty_interval/2 287
empty_queue/1 211
ensure_loaded/1 ..... 53, 75
entailed/1 249
environ/2 215
erase/1 115
error_exception/1 69
exec/3 215
execution_time (profile_data/4 option) 122
existence_error/5 (error class) 103
exp/1, exponent 96
exp/2, exponent 96
expand/0 266
expand_term/2 79

F

fail (debugger command) 66
fail/0 100
false/0 100
fcompile/1 ..... 52, 75
fd_closure/2 286
fd_dom/2 286
fd_global/3 286
fd_max/2 286
fd_min/2 286
fd_set/2 286
fd_size/2 286
fd_statistics/0 283
fd_statistics/2 282
fdset_add_element/3 288
fdset_complement/2 288
fdset_del_element/3 288
fdset_disjoint/2 288
fdset_eq/2 288
fdset_intersect/2 288
fdset_intersection/2 288
fdset_intersection/3 288
fdset_interval/3 287
fdset_max/2 287
fdset_member/2 288
fdset_min/2 287
fdset_parts/4 287
fdset_singleton/2 287
fdset_size/2 287
fdset_subset/2 288
fdset_subtract/3 288
fdset_to_list/2 287
fdset_to_range/2 288
fdset_union/2 288
fdset_union/3 288
file_exists/1 216
file_exists/2 216
file_property/2 216
file_search_path/2 89
fileerrors (prolog flag) 106
fileerrors/0 92
find this (debugger command) 67
find_constraint/2 313
find_constraint/3 313
findall/3 119
findall/4 119
findall_constraints/2 313
findall_constraints/3 313
float/1 110
float/1, coercion 94
floor/1, function 95
flush_output/0 91
flush_output/1 91
foreign/2 129
foreign/3 ..... 129, 168
foreign_file/2 134
foreign_resource/2 129
format/2 83
format/3 83
format_to_chars/3 369

format_to_chars/4 369
freeze/2 119
frozen/2 120
functor/3 111

G

garbage_collect/0 125
garbage_collect_atoms/0 125
garbage_collection (statistics/2 option) 109
gc (prolog flag) 106
gc/0 125
gc_margin (prolog flag) 106
gc_trace (prolog flag) 106
gcd/2, greatest common divisor 95
gen_assoc/3 191
gen_label/3 219
generate_flinkage/1 373
get/1 88
get/2 88
get_assoc/3 191
get_assoc/5 191
get_atts/2 193
get_default_db/1 237
get_from_heap/4 199
get_label/3 219
get_mutable/2 113
get_next_assoc/4 191
get_prev_assoc/4 192
get0/1 87
get0/2 87
getrand/1 213
global_stack (statistics/2 option) 108
GLOBALSTKSIZE (environment) 10
goal_expansion/3 79
ground/1 110

H

halt/0 124
halt/1 124
heap (statistics/2 option) 109
heap_size/2 199
heap_to_list/2 199
help (debugger command) 68
help/0 126
host_id/1 216
host_name/1 216
hostname_address/2 230

I

if/3 100
in/1 233
in/2 ..... 233, 276
in_noblock/1 233
in_set/2 276
incore/1 100
independent_set/3 223
indomain/1 280
inf/0, infinity 96
inf/2 250
initialization/0 126
initialization/1 126
insert_constraint/2 313
insert_constraint/3 313
instance/2 115
instantiation_error/2 (error class) 103
integer/1 110
integer/1, coercion 94
is/2 96
is_array/1 189
is_assoc/1 191
is_fdset/1 287
is_heap/1 199
is_list/1 201
is_mutable/1 ..... 110, 113
is_ordset/1 207
is_queue/1 211

K

keysort/2 98
kill/2 216

L

labeling/1 242
labeling/2 280
last/2 201
LD_LIBRARY_PATH (environment) 9, 155, 156
LD_RUN_PATH (environment) 9
leap (debugger command) 65
leash/1 ..... 62, 121
length/2 124
library_directory/1 90
LIBRARY_PATH (environment) 9
linda/0 232
linda/1 232
linda_client/1 233
linda_timeout/2 233
line_count/2 91
line_position/2 91

link_foreign_resource/6 133
list_queue/2 212
list_to_assoc/2 192
list_to_fdset/2 287
list_to_heap/2 199
list_to_ord_set/2 207
list_to_tree/2 219
listing/0 104
listing/1 104
load/1 ..... 53, 74
load_files/1 73
load_files/2 73
load_foreign_files/2 134
load_foreign_resource/1 133
local_stack (statistics/2 option) 109
LOCALSTKSIZE (environment) 10
log/1, logarithm 96
log/2, logarithm 96

M

make_directory/1 215
make_index:make_library_index/1 188
map_assoc/2 192
map_assoc/3 192
map_tree/3 219
max/2, maximum value 95
max_assoc/3 191
max_list/2 201
max_path/5 ..... 222, 226
max_workers (muse_flag option) 123
maximize/1 250
maximize/2 280
member/2 201
memberchk/2 201
memory (statistics/2 option) 109
meta_predicate/1 (declaration) 50, 56
method/3 (Java method identifier) 168
method_expansion/3 329
min/2, minimum value 95
min_assoc/3 191
min_list/2 202
min_of_heap/3 200
min_of_heap/5 200
min_path/5 ..... 222, 226
min_paths/3 ..... 222, 226
min_tree/3 ..... 223, 227
minimize/1 250
minimize/2 280
mktemp/2 216
mod/2, integer remainder 94
mode/1 (declaration) 56
module/1 105
module/2 (declaration) 48, 56
module/3 (declaration) 48, 56
msb/1, most significant bit 95
multifile/1 (declaration) 54
muse_flag/2 123
muse_flag/3 123
muse_flags/0 123
muse_init_lock() (C function) 150
muse_lock() (C function) 150
muse_max_workers() (C function) 150
muse_num_workers() (C function) 150
muse_sync/0 123
muse_trace/1 124
muse_trace/2 124
muse_un_lock() (C function) 150
muse_worker_id() (C function) 150

N

name/2 111
nan/0, not-a-number 96
neighbors/3 ..... 222, 226
neighbours/3 ..... 222, 226
new_array/1 189
nextto/3 202
nl/0 87
nl/1 87
no_doubles/1 202
nodebug (debugger command) 67
nodebug/0 ..... 62, 121
noexpand/0 266
nofileerrors/0 92
nogc/0 125
non_member/2 202
nonvar/1 110
nospy this (debugger command) 67
nospy/1 ..... 63, 121
nospyall/0 ..... 63, 121
notify_constrained/1 314
notrace/0 ..... 62, 121
nozip/0 ..... 62, 121
nth/3 202
nth/4 202
nth0/3 202
nth0/4 202
num_workers (muse_flag option) 123
number/1 110
number_chars/2 112
number_to_chars/2 369

number_to_chars/3 369
numbervars/3 124

O

on_exception/3 101
op/3 ..... 37, 124, 403
open/3 89
open/4 89
open_chars_stream/2 369
open_null_stream/1 91
ord_add_element/3 207
ord_del_element/3 207
ord_disjoint/2 207
ord_intersect/2 207
ord_intersection/2 207
ord_intersection/3 207
ord_intersection/4 207
ord_list_to_assoc/2 192
ord_member/2 208
ord_seteq/2 208
ord_setproduct/3 208
ord_subset/2 208
ord_subtract/3 208
ord_symdiff/3 208
ord_union/2 209
ord_union/3 208
ord_union/4 208
order_resource/2 282
ordering/1 ..... 251, 260
otherwise/0 100
out (debugger command) 66
out/1 233

P

parallel/0 (declaration) 57
parallel/1 (declaration) 57
path/3 ..... 222, 226
peek_char/1 88
peek_char/2 88
permission_error/5 (error class) 103
permutation/2 203
phrase/2 80
phrase/3 80
pid/1 216
popen/3 217
portray/1 82
portray/1 269
portray_clause/1 83
portray_clause/2 83
portray_message/2 102
predicate_property/2 104
prefix/2 203
prepare_foreign_resource/3 374
prepare_resource_table/2 373
print (debugger command) 66
print/1 82
print/2 82
print_message/2 102
profile_data/4 122
profile_reset/1 122
program (statistics/2 option) 109
project_attributes/2 195
prolog_flag/2 108
prolog_flag/3 105
prolog_load_context/2 108
PROLOGINCSIZE (environment) 10
PROLOGINITSIZE (environment) 10
PROLOGKEEPSIZE (environment) 10
PROLOGMAXSIZE (environment) 10
prompt/2 125
public/1 (declaration) 56
put/1 88
put/2 88
put_assoc/4 192
put_atts/2 193
put_label/4 219
put_label/5 219

Q

quasi-skip (debugger command) 66
queue/2 211
queue_head/3 211
queue_head_list/3 211
queue_last/3 211
queue_last_list/3 211
queue_length/2 212

R

raise exception (debugger command) 68
raise_exception/1 101
random/1 213
random/3 213
random_ugraph/3 223
random_wgraph/4 227
randseq/3 213
randset/3 213
range_to_fdset/2 287
rd/1 233
rd/2 234
rd_noblock/1 233

reachable/3 ..... 223, 227
read/1 80
read/2 80
read_from_chars/2 369
read_term/2 81
read_term/3 81
reconsult/1 74
recorda/3 116
recorded/3 116
recordz/3 116
redefine_warnings (prolog flag) 106
reduce/2 ..... 222, 226
reinitialise/0 125
relation/3 277
remove_constraint/1 314
remove_duplicates/2 203
rename_file/2 217
repeat/0 100
representation_error/3 (error class) 103
require/1 75
reset printdepth (debugger command) 68
reset subterm (debugger command) 68
restore/1 ..... 17, 125
retract/1 114
retractall/1 115
retry (debugger command) 66
reverse/2 203
round/1, function 95
runtime (statistics/2 option) 109

S

same_length/2 203
same_length/3 203
sat/1 242
save_program/1 ..... 17, 125
save_program/2 ..... 17, 125
scalar_product/4 276
see/1 93
seeing/1 93
seek/4 91
seen/0 93
select/3 203
sequential/0 (declaration) 57
sequential/1 (declaration) 57
serialized/2 278
serialized_precedence/3 279
serialized_precedence_resource/4 279
serialized_resource/3 279
set printdepth (debugger command) 68
set subterm (debugger command) 68
set_default_db/1 237
set_input/1 91
set_output/1 91
set_stream_position/2 91
setof/3 118
setrand/1 213
shallow_fails (profile_data/4 option) 122
shell/0 217
shell/1 217
shell/2 217
SHLIB_PATH (environment) 9
simple/1 110
sin/1, function 95
single_var_warnings (prolog flag) 106
sinh/1, function 95
skip (debugger command) 66
skip/1 88
skip/2 88
skip_line/0 88
skip_line/1 88
sleep/1 217
socket/2 229
socket_accept/2 229
socket_accept/3 229
socket_bind/2 229
socket_close/1 229
socket_connect/3 229
socket_listen/2 229
socket_select/5 230
socket_select/6 230
sort/2 98
source_file/1 75
source_file/2 75
source_info (prolog flag) 108
SP_ATOM (C macro) 139
SP_atom_from_string() (C function) 136
SP_atom_length() (C function) 136
SP_close_query() (C function) 142
SP_compare() (C function) 140
SP_COMPOUND (C macro) 139
SP_cons_functor() (C function) 137
SP_cons_list() (C function) 137
SP_continue() (C function) 143
SP_CTYPE (environment) 10
SP_cut_query() (C function) 142
SP_errno (C macro) 128
SP_ERROR (C macro) 128
SP_error_message() (C function) 128
SP_event() (C function) 142
SP_exception_term() (C function) 144

SP_FAILURE (C macro) 128
SP_fclose() (C function) 145
SP_fflush() (C function) 145
SP_fgetc() (C function) 145
SP_FLOAT (C macro) 139
SP_force_interactive() (C function) 151
SP_fprintf() (C function) 145
SP_fputc() (C function) 145
SP_fputs() (C function) 145
SP_free() (C function) 140
SP_get_address() (C function) 138
SP_get_arg() (C function) 138
SP_get_atom() (C function) 138
SP_get_float() (C function) 138
SP_get_functor() (C function) 138
SP_get_integer() (C function) 138
SP_get_list() (C function) 138
SP_get_list_chars() (C function) 138
SP_get_list_n_chars() (C function) 138
SP_get_number_chars() (C function) 138
SP_get_string() (C function) 138
SP_getc() (C function) 145
SP_initialize() (C function) 151
SP_INTEGER (C macro) 139
SP_is_atom() (C function) 139
SP_is_atomic() (C function) 139
SP_is_compound() (C function) 139
SP_is_float() (C function) 139
SP_is_integer() (C function) 139
SP_is_list() (C function) 139
SP_is_number() (C function) 139
SP_is_variable() (C function) 139
SP_load() (C function) 153
SP_make_stream() (C function) 147
SP_malloc() (C function) 140
SP_new_term_ref() (C function) 136
SP_next_solution() (C function) 142
SP_on_fault() (C macro) 153
SP_open_query() (C function) 142
SP_PATH (environment) 10, 127
SP_pred() (C function) 141
SP_predicate() (C function) 141
SP_printf() (C function) 145
SP_put_address() (C function) 137
SP_put_atom() (C function) 137
SP_put_float() (C function) 137
SP_put_functor() (C function) 137
SP_put_integer() (C function) 137
SP_put_list() (C function) 137
SP_put_list_chars() (C function) 137
SP_put_list_n_chars() (C function) 137
SP_put_number_chars() (C function) 137
SP_put_string() (C function) 137
SP_put_term() (C function) 136
SP_put_variable() (C function) 137
SP_putc() (C function) 145
SP_puts() (C function) 145
SP_qid (C type) 142
SP_query() (C function) 141
SP_query_cut_fail() (C function) 141
SP_raise_exception() (C function) 144
SP_raise_fault() (C function) 154
SP_realloc() (C function) 140
SP_register_atom() (C function) 137
SP_reinstall_signal() (C function) 143
SP_restore() (C function) 153
SP_set_interrupt_hook (C function) 150
SP_set_memalloc_hooks() (C function) 152
SP_set_read_hook (C function) 150
SP_set_reinit_hook (C function) 150
SP_set_tty() (C function) 147
SP_set_user_stream_hook() (C function) 148
SP_set_user_stream_post_hook() (C function) ... 148
SP_signal() (C function) 143
SP_stream (C type) 145
SP_string_from_atom() (C function) 136
SP_SUCCESS (C macro) 128
SP_term_ref (C type) 127
SP_term_type() (C function) 139
SP_unify() (C function) 140
SP_unregister_atom() (C function) 137
SP_VARIABLE (C macro) 139
splfr (utility) 133
spmkds (utility) 156
spmkrs (utility) 154
spy this (debugger command) 67
spy/1 ..... 62, 121
spypoint_condition/3 63
sqrt/1, square root 96
stack_shifts (statistics/2 option) 109
statistics/0 108
statistics/2 108
stream_code/2 145
stream_interrupt/3 92
stream_position/2 91
stream_select/3 92
sublist/2 203
substitute/4 203
subsumes/2 205
subsumes_chk/2 205

suffix/2 203
sum/3 276
sum_list/2 203
sup/2 250
symmetric_closure/2 ..... 222, 226
syntax_error/5 (error class) 103
syntax_errors (prolog flag) 107
system/0 217
system/1 217
system/2 217
system_error/1 (error class) 103
system_type (prolog flag) 107

T

tab/1 88
tab/2 88
tan/1, function 95
tanh/1, function 95
taut/2 242
tcl_delete/1 361
tcl_eval/3 361
tcl_new/1 361
tell/1 93
telling/1 93
term_expansion/2 79
term_expansion/4 79
term_hash/2 206
term_hash/4 206
term_subsumer/3 205
term_variables/2 206
time_out/3 375
tk_all_events (tk_do_one_event/1 option) 365
tk_destroy_window/1 365
tk_do_one_event/0 365
tk_do_one_event/1 365
tk_dont_wait (tk_do_one_event/1 option) 365
tk_file_events (tk_do_one_event/1 option) 365
tk_idle_events (tk_do_one_event/1 option) 365
tk_main_loop/0 366
tk_main_window/1 365
tk_make_window_exist/1 365
tk_new/2 364
tk_next_event/2 366
tk_next_event/3 366
tk_num_main_windows/1 365
tk_timer_events (tk_do_one_event/1 option) 365
tk_window_events (tk_do_one_event/1 option) 365
tk_x_events (tk_do_one_event/1 option) 365
TMPDIR (environment) 10, 124
tmpnam/1 217
told/0 93
top_sort/2 ..... 222, 226
toplevel_print_options (prolog flag) 107
trace/0 ..... 61, 121
trail (statistics/2 option) 109
TRAILSTKSIZE (environment) 10
transitive_closure/2 ..... 222, 226
transpose/2 ..... 222, 226
tree_size/2 219
tree_to_list/2 219
true/0 100
truncate/1, function 95
ttyflush/0 88
ttyget/1 88
ttyget0/1 88
ttynl/0 88
ttyput/1 88
ttyskip/1 88
ttytab/1 88
type_error/4 (error class) 103
typein_module (prolog flag) 107

U

ugraph_to_wgraph/2 225
unconstrained/1 314
unify (debugger command) 68
unify_with_occurs_check/2 206
unknown (prolog flag) 106
unknown/2 ..... 15, 120
unknown_predicate_handler/3 103
unload_foreign_resource/1 133
update_mutable/2 113
use_module/1 75
use_module/2 75
use_module/3 75
user:error_exception/1 69
user:file_search_path/2 89
user:goal_expansion/3 79
user:library_directory/1 90
user:method_expansion/3 329
user:portray/1 82
user:portray_message/2 102
user:term_expansion/2 79
user:term_expansion/4 79
user:unknown_predicate_handler/3 103
user:user_help/0 126
user_error (prolog flag) 107
user_error (stream alias) 72
user_help/0 126
user_input (prolog flag) 107

user_input (stream alias) 72
user_output (prolog flag) 107
user_output (stream alias) 72

V

var/1 110
variant/2 205
verify_attributes/3 194
version (muse_flag option) 123
version (prolog flag) 107
version/0 126
version/1 126
vertices/2 ..... 221, 225
vertices_edges_to_ugraph/3 221
vertices_edges_to_wgraph/3 225
view/1 367
volatile/1 (declaration) 55

W

wait/2 217
walltime (statistics/2 option) 109
wgraph_to_ugraph/2 225
when/2 119
with_output_to_chars/2 370
with_output_to_chars/3 370
with_output_to_chars/4 370
worker_id (muse_flag option) 123
working_directory/2 217
write (debugger command) 67
write/1 82
write/2 82
write_canonical/1 82
write_canonical/2 82
write_term/2 83
write_term/3 83
write_to_chars/2 369
write_to_chars/3 369
writeq/1 82
writeq/2 82

X

xref/1 377

Z

zip (debugger command) 65
zip/0 ..... 61, 121

Prolog Objects Method Index

A

abolish/0 (object method) 343
ancestor/1 (utility method) 344
ancestor/2 (utility method) 344
ancestors/1 (utility method) 344
ancestors/2 (utility method) 344
and_cast/2 (utility method) 344
assert/1 (object method) 339, 343
assert/2 (object method) 343
asserta/1 (object method) 339, 343
asserta/2 (object method) 343
assertz/1 (object method) 339, 343
assertz/2 (object method) 343
attributes/1 (universal method) 341
augment/1 (object method) 343
augmenta/1 (object method) 343
augmentz/1 (object method) 343

D

descendant/1 (utility method) 344
descendant/2 (utility method) 344
descendants/1 (utility method) 344
descendants/2 (utility method) 344
dynamic/0 (object method) 336, 342
dynamic/1 (object method) 337, 342
dynamic_methods/1 (utility method) 344
dynamic_objects/1 (utility method) 344

G

get/1 (inlined method) 341
get/1 (object method) 342

H

has_attribute/1 (object method) 342
has_instance/1 (object method) 342

I

instance/1 (object method) 342

M

methods/1 (utility method) 344

N

new/1 (object method) 339, 342
new/2 (object method) 339, 342

O

object (built-in object) 342
object/1 (object method) 342
objects/1 (utility method) 343
or_cast/2 (utility method) 344

P

pltrace-port-arrow-assoc 22
prolog-align-comments-flag 23
prolog-electric-dot-flag 23
prolog-electric-newline-flag 23
prolog-electric-underscore-flag 23
prolog-hungry-delete-key-flag 23
prolog-imenu-flag 24
prolog-indent-mline-comments-flag 23
prolog-indent-width 22
prolog-info-predicate-index 24
prolog-keywords 23
prolog-object-end-to-0-flag 23
prolog-old-sicstus-keys-flag 23
prolog-paren-indent 22
prolog-parse-mode 23
prolog-system 22
prolog-system-version 22
prolog-underscore-wordchar-flag 24
prolog-use-prolog-tokenizer-flag 23
prolog-use-sicstus-sd 22

R

restart/0 (utility method) 344
retract/1 (object method) 343
retractall/1 (object method) 343

S

self/1 (inlined method) 341
self/1 (object method) 342
set/1 (inlined method) 341
set/1 (object method) 342
static/0 (object method) 342
static/1 (object method) 342
static_methods/1 (utility method) 344

static_objects/1 (utility method) 344
sub/1 342
sub/1 (object method) 335
subs/1 (utility method) 343
super (keyword) 336
super/1 342
super/2 (universal method) 334, 341
supers/1 (utility method) 343

U

update/1 (object method) 343
utility (built-in object) 343

Concept Index

A

abort ... 17
access driven programming ... 340
alias, file name ... 72, 89
all solutions ... 117
ancestor goal ... 64
anonymous variable ... 28
ANSI conformance ... 6, 127
anti-unifications ... 205
API ... 127
arguments, command-line ... 7
arithmetic ... 94
arity ... 28
arrays ... 189
assignment, destructive ... 113
association lists ... 191
asynchronous predicate ... 46
asynchronously, calling Prolog ... 142
atom ... 28
attribute declaration ... 193
attributed variables ... 193
attributes, object ... 327
attributes, object, implementation ... 345

B

backtracking ... 35
binary trees ... 219
blackboard ... 46, 116
block declaration ... 55
blocked goal ... 64
body ... 30
break ... 16
built-in predicate ... 32

C

call, last ... 178
calling Prolog asynchronously ... 142
calling Prolog from C ... 141
cavalier predicate ... 46
character I/O ... 87
character set ... 10, 398
choicepoint ... 43
CHR control flow model ... 317
CHR debugging messages ... 320
CHR debugging options ... 320
CHR debugging predicates ... 317
CHR spy-points ... 319
clause ... 30
clause, guarded ... 179
command ... 8
command-line arguments ... 7
communication ... 231
comparing terms ... 97
compilation ... 73
compiling ... 7
compound term ... 27, 28
computation rule ... 35
conformance, ANSI ... 6, 127
considerations for fcompile ... 58
consistent store ... 283
constant ... 27
constraint ... 272
constraint store ... 283
constraint, global ... 285
consulting ... 7, 11, 52, 73
context, Directory ... 108
context, File ... 108
context, Module ... 108
contradictory store ... 283
conversions, term ... 135
coroutining ... 119
counter ... 121
cross reference ... 377
current input stream ... 72
current output stream ... 72
cut ... 36, 44
cut, green ... 37
cut, red ... 37
cyclic term ... 36, 81, 83

D

daemon ... 340
database ... 115, 235
database, default ... 237
db-spec ... 236
DCG ... 76
debug options ... 65
debugging ... 59
debugging messages ... 63
debugging predicates ... 61

declaration, attribute 193
declaration, block 55
declaration, dynamic 54
declaration, meta-predicate 50, 56
declaration, mode 56
declaration, module 56
declaration, multifile 52, 54
declaration, operator 53
declaration, predicate 53
declaration, public 56
declaration, volatile 55
declarations 53
declarative semantics 33
deep failure 122, 178
default database 237
definite clause 27
delegation 327
delegation, message 332
destructive assignment 113
development kernel 373
development system 6, 156
differential inheritance 335
directive 8, 12
domain variable 272
domain, finite 272
domain-consistent 284
domain-disentailed 284
domain-entailed 284
dynamic declaration 54
dynamic linking 129
dynamic method 337
dynamic object 327, 336, 338
dynamic predicate 54

E

emacs interface 18
end of file 5
end of line 92
end of stream 91
EOF 5
error handling 69, 101
error, syntax 14
escape sequence 106, 401
exception handling 69, 101
exception handling in C 144
execution 15
execution profiling 121
execution, nested 16
exiting 16
expansion, macro 79
expansion, module name 49, 56
exported predicate 47
external database 235
external storage 235

F

failure, deep 122, 178
failure, shallow 122, 178
fcompile, considerations for 58
FD predicate 293
FD set 287
file 71
file name alias 72, 89
file, module 48
filename 72
Finding Libraries 9
Finding Shared Objects 9
finite domain 272
flag, Prolog 105, 108
float 27
floundering 35
foreign language interface 127, 128
foreign resource 128
foreign resource, linked 128, 135
function prototype 6
functor 28

G

garbage collection 106
gauge 367
GCLA 359
generalized Horn clause language 359
generic object 331
generic object, implementation 348
global constraint 285
goal 30
goal, ancestor 64
goal, blocked 64
grammar rule 76
graphical tracing 44
graphs, unweighted 221
graphs, weighted 225
green cut 37
guarded clause 179

H

handling, interrupt 142
handling, signal 142
head 30
heaps 199
hidden module 48
hierarchy, object 333
hook functions for I/O 150
hook functions for reinitialization 150
hook predicate 54
Horn clause 27

I

I/O hook functions 150
if-then-else 178
implicit parallelism 41
importation 49
imported predicate 47
indexed term 236
indexical 291
indexing 178
information, source 19, 108
inheritance 327, 333, 334
inheritance by overriding 327
inheritance, differential 335
inheritance, multiple 334
input 71
input stream, current 72
instance variable 351
instances 341
instances, implementation 346
integer 27
interface, emacs 18
interface, foreign language 127, 128
interoperability 127
interrupt handling 142
interrupt, stream 92
interval-consistent 284
interval-disentailed 284
interval-entailed 284

K

kernel, development 373
kernel, runtime 373
keyboard 5

L

last call 178
layout term 81
library 187
Linda 231
line, end of 92
linked foreign resource 128, 135
linking, dynamic 129
linking, static 129, 154
list 29
lists 201
loading 51, 73
logic programming 1

M

macro expansion 79
message delegation 332
message sending 332
messages, suppressing 101
meta-logical predicate 110
meta-predicate declaration 50, 56
method 327
method, dynamic 337
mixing C and Prolog 127
mode declaration 56
mode spec 5
module declaration 56
module file 48
module name expansion 49, 56
module system 47
module, hidden 48
module, object 345
module, source 47
module, type-in 47, 107
multifile declaration 52, 54
multiple inheritance 334
Muse model 43
Must tracing 45
mutable term 113

N

nested execution 16
non-unit clause 31
notation 5

O

object ... 327
object hierarchy ... 333
object module ... 345
object, dynamic ... 327, 336, 338
object, generic ... 331
object, generic, implementation ... 348
object, parameterized ... 331
object, parameterized, implementation ... 348
object, static ... 327, 336
object-oriented programming ... 327
occurs-check ... 36, 205
operating system ... 215
operator declaration ... 53
operators ... 37, 124, 403
ordered sets ... 207
output ... 71
output stream, current ... 72
overriding, inheritance by ... 327

P

parallel declaration ... 57
parallelism, implicit ... 41
parameterized object ... 331
parameterized object, implementation ... 348
post, to ... 273
predicate ... 30, 32
predicate declaration ... 53
predicate spec ... 5
predicate, asynchronous ... 46
predicate, cavalier ... 46
predicate, dynamic ... 54
predicate, exported ... 47
predicate, FD ... 293
predicate, hook ... 54
predicate, imported ... 47
predicate, private ... 47
predicate, public ... 47
predicate, undefined ... 15, 106
predicate, volatile ... 55
priority queues ... 199
private predicate ... 47
procedural semantics ... 34
procedure box ... 59
procedure call ... 35
procedure definition ... 34
process communication ... 229, 231
profiling ... 367
profiling, execution ... 121
program ... 30
program state ... 104
programming in logic ... 1
programming, access driven ... 340
programming, object-oriented ... 327
Prolog flag ... 105, 108
prototype ... 327
prototype, function ... 6
public declaration ... 56
public predicate ... 47

Q

query ... 8
queues ... 211

R

random numbers ... 213
reading in ... 11
reconsult ... 68
recursion, tail ... 178
red cut ... 37
reference, term ... 127
reification ... 275
reinitialization ... 125, 126
reinitialization hook functions ... 150
repeat loop ... 93
resource, foreign ... 128
restoring ... 17
rule, computation ... 35
rule, grammar ... 76
rule, search ... 35
running ... 7
runtime kernel ... 373
runtime system ... 6, 151

S

saved state ... 17
saving ... 17
scheduler ... 43
scheduling ... 43
search rule ... 35
self ... 332
semantics ... 33, 34
sending, message ... 332
sentence ... 30
sequence, escape ... 106, 401
sequential declaration ... 57
sets ... 207
shallow failure ... 122, 178
side effects ... 44
signal handling ... 142

sockets ... 229
solutions, all ... 117
source information ... 19, 108
source module ... 47
SP_term_ref (C type) ... 136
speculative work ... 44
spy-point ... 62
stand-alone application ... 151
standard order ... 97
state, program ... 104
state, saved ... 17
static linking ... 129, 154
static object ... 327, 336
store, consistent ... 283
store, constraint ... 283
store, contradictory ... 283
stream ... 71
stream, end of ... 91
string ... 30
subsumption ... 205
suppressing messages ... 101
suspension ... 44
suspension, voluntary ... 44
synchronization ... 231
synchronized ... 44
syntax error ... 14
system, development ... 6
system, operating ... 215
system, runtime ... 6

T

tail recursion ... 178
term ... 27
term comparison ... 97
term conversions ... 135
term I/O ... 80
term reference ... 127
term, compound ... 28
term, cyclic ... 36, 81, 83
term, indexed ... 236
term, layout ... 81
term, mutable ... 113
terms ... 205
top level ... 8
tracing, graphical ... 44
trees ... 219
type-in module ... 47, 107

U

ugraph ... 221
undefined predicate ... 15, 103, 106
unification ... 35
unit clause ... 30
unweighted graphs ... 221
user ... 12

V

variable ... 27, 28
variable, domain ... 272
variable, instance ... 351
variables, attributed ... 193
Visandor tracing ... 45
volatile declaration ... 55
volatile predicate ... 55
voluntary suspension ... 44

W

WAM ... 1, 43
weighted graphs ... 225
wgraph ... 225
work ... 43
work, speculative ... 44
worker ... 43

Table of Contents

Introduction 1

Acknowledgments 3

Notational Conventions 5

Keyboard Characters 5
Mode Spec 5
Predicate Spec 5
Development and Runtime Systems 6
Function Prototypes 6
ISO Compliance 6

1 How to Run Prolog 7

1.1 Getting Started 7
1.1.1 Finding Libraries at Runtime 9
1.1.2 Environment Variables 10
1.2 Reading in Programs 11
1.3 Inserting Clauses at the Terminal 12
1.4 Directives: Queries and Commands 12
1.4.1 Queries 12
1.4.2 Commands 13
1.5 Syntax Errors 14
1.6 Undefined Predicates 15
1.7 Program Execution And Interruption 15
1.8 Exiting From The Top-Level 16
1.9 Nested Executions—Break and Abort 16
1.10 Saving and Restoring Program States 17
1.11 Emacs Interface 18
1.11.1 Customizing Emacs 18
1.11.2 Basic Configuration 19
1.11.3 Usage 19
1.11.4 Mode Line 21
1.11.5 Configuration 22
1.11.6 Tips 24
1.11.6.1 Font-locking 24
1.11.6.2 Auto-fill mode 24
1.11.6.3 Speed 24
1.11.6.4 Changing Colors 25

2 The Prolog Language 27

2.1 Syntax, Terminology and Informal Semantics 27
2.1.1 Terms 27
2.1.1.1 Integers 27
2.1.1.2 Floats 27
2.1.1.3 Atoms 28
2.1.1.4 Variables 28
2.1.1.5 Compound Terms 28
2.1.2 Programs 30
2.2 Declarative Semantics 33
2.3 Procedural Semantics 34
2.4 Occurs-Check 36
2.5 The Cut Symbol 36
2.6 Operators 37
2.7 Syntax Restrictions 40
2.8 Comments 40

3 Running Prolog in Parallel 41

3.1 Introduction 41
3.2 Running a Parallel Example 41
3.3 Main Issues of Parallel Execution 43
3.3.1 The Muse Model 43
3.3.2 Scheduling 43
3.3.3 Cut, Side Effects and Suspension 44
3.4 Visualization Tools 44
3.4.1 Must 45
3.4.2 Visandor 45
3.4.3 Vimust 45
3.5 Programming Considerations 45

4 The Module System 47

4.1 Basic Concepts 47
4.2 Module Prefixing 48
4.3 Defining Modules 48
4.4 Importation 49
4.5 Module Name Expansion 49
4.6 Meta-Predicate Declarations 50

5 Loading Programs 51

5.1 Predicates which Load Code 52
5.2 Declarations 53
5.2.1 Multifile Declarations 54
5.2.2 Dynamic Declarations 54
5.2.3 Volatile Declarations 55
5.2.4 Block Declarations 55
5.2.5 Meta-Predicate Declarations 56
5.2.6 Module Declarations 56
5.2.7 Public Declarations 56

5.2.8 Mode Declarations . . . 56
5.2.9 Parallel and Sequential Declarations . . . 57
5.3 Considerations for File-To-File Compilation . . . 58

6 Debugging . . . 59
6.1 The Procedure Box Control Flow Model . . . 59
6.2 Basic Debugging Predicates . . . 61
6.3 Spy-points . . . 62
6.4 Format of Debugging messages . . . 63
6.5 Options Available during Debugging . . . 65
6.6 Consulting during Debugging . . . 68
6.7 Catching Exceptions . . . 69

7 Built-In Predicates . . . 71
7.1 Input / Output . . . 71
7.1.1 Reading-in Programs . . . 73
7.1.2 Term and Goal Expansion . . . 76
7.1.3 Input and Output of Terms . . . 80
7.1.4 Character Input/Output . . . 87
7.1.5 Stream I/O . . . 89
7.1.6 DEC-10 Prolog File I/O . . . 93
7.1.7 An Example . . . 93
7.2 Arithmetic . . . 94
7.3 Comparison of Terms . . . 97
7.4 Control . . . 99
7.5 Error and Exception Handling . . . 101
7.6 Information about the State of the Program . . . 104
7.7 Meta-Logic . . . 110
7.8 Modification of Terms . . . 113
7.9 Modification of the Program . . . 113
7.10 Internal Database . . . 115
7.11 Blackboard Primitives . . . 116
7.12 All Solutions . . . 117
7.13 Coroutining . . . 119
7.14 Debugging . . . 120
7.15 Execution Profiling . . . 121
7.16 Muse . . . 123
7.17 Miscellaneous . . . 124

8 Mixing C and Prolog . . . 127
8.1 Notes . . . 127
8.2 Calling C from Prolog . . . 128
8.2.1 Foreign Resource and Conversion Declarations . . . 128
8.2.2 Static and Dynamic linking . . . 129
8.2.3 Conversion Declarations . . . 129
8.2.4 Conversions between Prolog Arguments and C Types . . . 130
8.2.5 Interface Predicates . . . 133
8.2.6 Init and Deinit Functions . . . 134

8.2.7 Creating the Linked Foreign Resource 135
8.3 Support Functions 135
8.3.1 Creating and Manipulating SP_term_refs 136
8.3.2 Creating Prolog Terms 137
8.3.3 Accessing Prolog Terms 138
8.3.4 Testing Prolog Terms 139
8.3.5 Unifying and Comparing Terms 140
8.3.6 Memory Allocation 140
8.4 Calling Prolog from C 141
8.4.1 Finding One Solution of a Call 141
8.4.2 Finding Multiple Solutions of a Call 141
8.4.3 Calling Prolog Asynchronously 142
8.4.4 Exception Handling in C 144
8.5 SICStus Streams 144
8.5.1 Prolog Streams 145
8.5.2 Defining a New Stream 146
8.5.2.1 Low Level I/O Functions 146
8.5.2.2 Installing a New Stream 147
8.5.2.3 Internal Representation 147
8.5.3 Hookable Standard Streams 148
8.5.3.1 Writing User-stream Hooks 148
8.5.3.2 Writing User-stream Post-hooks 149
8.5.3.3 User-stream Hook Example 149
8.6 Muse Support Functions 150
8.7 Hooks 150
8.8 Runtime Systems 151
8.8.1 Initializing the Prolog Engine 151
8.8.2 Loading Prolog Code 153
8.8.3 Creating the Executable 154
8.8.4 Running the Executable 155
8.9 Development Systems 156
8.10 Examples 156
8.10.1 Train Example (connections) 157
8.10.2 I/O on Lists of Character Codes 160
8.10.3 Exceptions from C 162
8.10.4 Stream Example 164

9 Mixing Java and Prolog 167

9.1 Prerequisites 167
9.2 Calling Java from Prolog 167
9.2.1 Static and Dynamic Linking 168
9.2.2 Declarating Java-methods 168
9.3 Conversions between Prolog Arguments and Java Types 168
9.3.1 Calling Java from Prolog: An Example 172
9.4 Calling Prolog from Java 173
9.5 Jasper Package Class Reference 175
9.6 Exception Handling 176
9.7 Resources 176

10 Programming Tips and Examples 177
10.1 Programming Guidelines 177
10.2 Indexing 178
10.3 Last Call Optimization 178
10.4 If-Then-Else Compilation 178
10.5 Programming Examples 179
10.5.1 Simple List Processing 179
10.5.2 Family Example (descendants) 179
10.5.3 Association List Primitives 180
10.5.4 Differentiation 181
10.5.5 Use of Meta-Logical Predicates 181
10.5.6 Use of Term Expansion 182
10.5.7 Prolog in Prolog 182
10.5.8 Translating English Sentences into Logic Formulae 183
10.5.9 Muse FLI Example 184

11 The Prolog Library 187

12 Array Operations 189

13 Association Lists 191

14 Attributed Variables 193

15 Heap Operations 199

16 List Operations 201

17 Term Utilities 205

18 Ordered Set Operations 207

19 Queue Operations 211

20 Random Number Generator 213

21 Operating System Utilities 215

22 Updatable Binary Trees 219

23 Unweighted Graph Operations 221

24 Weighted Graph Operations 225

25 Socket I/O 229

26 Linda—Process Communication 231
26.1 Server 232
26.2 Client 233

27 External Storage of Terms (External Database) 235
27.1 Basics 235
27.2 Current Limitations 235
27.3 The DB-Spec—Informal Description 236
27.4 Predicates 236
27.5 An Example Session 238
27.6 The Db-Spec 239

28 Boolean Constraint Solver 241
28.1 Solver Interface 242
28.2 Examples 242
28.2.1 Example 1 242
28.2.2 Example 2 243
28.2.3 Example 3 243
28.2.4 Example 4 244

29 Constraint Logic Programming over Rationals or Reals 247
29.1 Introduction 247
Referencing this Software 247
Acknowledgments 247
29.2 Solver Interface 248
Notational Conventions 248
29.2.1 Solver Predicates 248
29.2.2 Unification 252
29.2.3 Feedback and Bindings 253
29.3 Linearity and Nonlinear Residues 253
29.3.1 How Nonlinear Residues are made to disappear 254
29.3.2 Isolation Axioms 254
29.4 Numerical Precision and Rationals 255
29.5 Projection and Redundancy Elimination 259
29.5.1 Variable Ordering 260
29.5.2 Turning Answers into Terms 261
29.5.3 Projecting Inequalities 261
29.6 Why Disequations 264
29.7 Syntactic Sugar 265
29.7.1 Monash Examples 266
29.7.1.1 Compatibility Notes 267
29.8 A Mixed Integer Linear Optimization Example 268
29.9 Implementation Architecture 269
29.9.1 Fragments and Bits 269
29.9.1.1 Rationals 269
29.9.1.2 Partial Evaluation, Compilation 270

29.9.1.3 Asserting with Constraints 270
29.9.2 Bugs 270

30 Constraint Logic Programming over Finite Domains 271

30.1 Introduction 271
Referencing this Software 271
Acknowledgments 272
30.2 Solver Interface 272
30.2.1 Posting Constraints 273
30.2.2 A Constraint Satisfaction Problem 273
30.2.3 Reified Constraints 274
30.3 Available Constraints 275
30.3.1 Arithmetic Constraints 275
30.3.2 Membership Constraints 276
30.3.3 Propositional Constraints 276
30.3.4 Combinatorial Constraints 277
30.3.5 User-Defined Constraints 280
30.4 Enumeration Predicates 280
30.5 Statistics Predicates 282
30.6 Answer Constraints 283
30.7 The Constraint System 283
30.7.1 Definitions 283
30.7.2 Pitfalls of Interval Reasoning 284
30.8 Defining Global Constraints 285
30.8.1 The Global Constraint Programming Interface 285
30.8.2 Reflection Predicates 286
30.8.3 FD Set Operations 287
30.8.4 A Global Constraint Example 288
30.9 Defining Primitive Constraints 290
30.9.1 Indexicals 291
30.9.2 Range Expressions 291
30.9.3 Term Expressions 292
30.9.4 Monotonicity of Indexicals 293
30.9.5 FD predicates 293
30.9.6 Execution of Propagating Indexicals 296
30.9.7 Execution of Checking Indexicals 296
30.9.8 Goal Expanded Constraints 297
30.10 Example Programs 298
30.10.1 Send More Money 298
30.10.2 N Queens 298
30.10.3 Cumulative Scheduling 300
30.11 Syntax Summary 301
Syntax of Indexicals 301
Syntax of Arithmetic Expressions 303
Operator Declarations 304

31 Constraint Handling Rules 305

Copyright 305
31.1 Introduction 305
31.2 Introductory Examples 306
31.3 CHR Library 308
31.3.1 Loading the Library 308
31.3.2 Declarations 308
31.3.3 Constraint Handling Rules, Syntax 309
31.3.4 How CHR work 310
31.3.5 Pragmas 311
31.3.6 Options 312
31.3.7 Built-In Predicates 313
31.3.8 Consulting and Compiling Constraint Handlers 314
31.3.9 Compiler-generated Predicates 314
31.3.10 Operator Declarations 315
31.3.11 Exceptions 315
31.4 Debugging CHR Programs 316
31.4.1 Control Flow Model 317
31.4.2 CHR Debugging Predicates 317
31.4.3 CHR Spy-points 319
31.4.4 CHR Debugging Messages 320
31.4.5 CHR Debugging Options 320
31.5 Programming Hints 323
31.6 Constraint Handlers 324
31.7 Backward Compatibility 326

32 Prolog Objects 327

32.1 Getting Started 328
32.2 Declared Objects 328
32.2.1 Object Declaration 328
32.2.2 Method Declarations 329
32.2.3 Generic Objects for Easy Reuse 331
32.3 Self, Message Sending, and Message Delegation 332
32.4 Object Hierarchies, Inheritance, and Modules 333
32.4.1 Inheritance 334
32.4.2 Differential Inheritance 335
32.4.3 Use of Modules 335
32.4.4 Super and Sub 335
32.4.5 The Keyword Super 336
32.4.6 Semantic Links to Other Objects 336
32.4.7 Dynamically Declared Objects 336
32.4.8 Dynamic Methods 337
32.4.9 Inheritance of Dynamic Behavior 338
32.5 Creating Objects Dynamically 338
32.5.1 Object Creation 339
32.5.2 Method Additions 339
32.5.3 Parameter Passing to New Objects 340
32.6 Access Driven Programming—Daemons 340
32.7 Instances 341

32.8 Built-In Objects and Methods ... 341
32.8.1 Universal Methods ... 341
32.8.2 Inlined Methods ... 341
32.8.3 The Proto-Object "object" ... 342
32.8.4 The built-in object "utility" ... 343
32.9 Expansion to Prolog Code ... 345
32.9.1 The Inheritance Mechanism ... 345
32.9.2 Object Attributes ... 345
32.9.3 Object Instances ... 346
32.9.4 The Object Declaration ... 346
32.9.5 The Method Code ... 346
32.9.6 Parameter Transfer ... 348
32.10 Examples ... 350
32.10.1 Classification of Birds ... 350
32.10.2 Inheritance and Delegation ... 351
32.10.3 Prolog++ programs ... 354

33 Generalized Horn Clause Language ... 359

34 Tcl/Tk Interface ... 361

34.1 Prolog to Tcl ... 361
34.1.1 Command Format ... 362
34.2 Tcl to Prolog ... 363
34.3 Tk ... 364

35 The Gauge Profiling Tool ... 367

36 I/O on Lists of Character Codes ... 369

37 Jasper ... 371

38 Glue Code Generator ... 373

39 Timeout Predicate ... 375

40 Cross Reference Producer ... 377

Summary of Built-In Predicates ... 379

Full Prolog Syntax ... 395

Notation ... 395
Syntax of Sentences as Terms ... 396
Syntax of Terms as Tokens ... 397
Syntax of Tokens as Character Strings ... 398
Escape Sequences ... 401
Notes ... 402

Standard Operators **403**

References **405**

Predicate Index **409**

Prolog Objects Method Index **421**

Concept Index **423**